UNIX™ System V User Guide

June 1984
307-100, Issue 3

UNIX is a trademark of AT&T Bell Laboratories

Copyright © 1984 AT&T Technologies, Inc.
All Rights Reserved
Printed in U.S.A.

AT&T

CONTENTS

HOW TO READ THIS GUIDE vii

PART 1. UNIX SYSTEM OVERVIEW

CHAPTER 1. WHAT IS THE UNIX SYSTEM?

What The UNIX System Is 1-1
How The UNIX System Works 1-3

CHAPTER 2. BASICS FOR UNIX SYSTEM USERS

Getting Started 2-1
About The Terminal 2-2
Obtaining A Login Name 2-11
Establishing Contact With The UNIX System 2-11

CHAPTER 3. USING THE FILE SYSTEM

Introduction 3-1
How The File System Is Structured 3-4
Your Place In The File System Structure 3-4
Organizing A Directory Structure 3-16
Accessing And Manipulating Files 3-29
Summary 3-64

CHAPTER 4. UNIX SYSTEM CAPABILITIES

Introduction 4-1
Text Editing 4-1
Working In The Shell 4-6
Communicating Electronically 4-20
Programming In The System 4-21

PART 2. UNIX SYSTEM TUTORIALS

CHAPTER 5. LINE EDITOR TUTORIAL (ed)

Introducing The Line Editor 5-1
How To Read This Tutorial 5-2
Getting Started 5-3
Exercise 1 5-13
General Format Of ed Commands 5-13
Line Addressing 5-14
Exercise 2 5-29
Display Lines In A File 5-30
Creating Text 5-33
Exercise 3 5-39
Deleting Text 5-41
Substituting Text 5-47
Exercise 4 5-54
Special Characters 5-56
Exercise 5 5-67
Moving Text 5-69
Exercise 6 5-79
Other Useful Commands And Information 5-79
Exercise 7 5-88
Answers to Exercises 5-90

CHAPTER 6. SCREEN EDITOR TUTORIAL (vi)

Getting Acquainted With vi 6-1
How To Read This Tutorial 6-2
Getting Started 6-5
Exercise 1 6-15
Positioning The Cursor In The Window 6-16
Positioning The Cursor In The File 6-34
Exercise 2 6-45
Creating Text 6-46
Exercise 3 6-50

CHAPTER 6. SCREEN EDITOR TUTORIAL (vi) (Continued)

Deleting Text 6-51
Exercise 4 6-60
Changing Text 6-60
Cutting And Pasting Text Electronically 6-66
Exercise 5 6-70
Special Commands 6-71
Line Editing Commands 6-74
Quitting vi 6-80
Special Options For vi 6-82
Exercise 6 6-84
Changing Your Environment 6-85
Answers to Exercises 6-88

CHAPTER 7. SHELL TUTORIAL

Making Life Easier In The Shell 7-1
How To Read This Tutorial 7-2
Shell Command Language 7-3
Command Language Exercises 7-31
Shell Programming 7-32
Shell Programming Exercises 7-86
Answers to Exercises 7-88

CHAPTER 8. COMMUNICATION TUTORIAL

Introduction 8-1
Communicating On The UNIX System 8-2
How Can You Communicate? 8-3
Sending And Receiving Messages 8-4
Sending And Receiving Files 8-17
Advanced Message And File Handling 8-29

PART 3. SUPPLEMENTARY INFORMATION AND REFERENCE TOOLS

Appendix A. Selected UNIX System Documentation A-1
Appendix B. File System Organization B-1
Appendix C. Summary of UNIX System Commands C-1
Appendix D. Quick Reference to ed Commands D-1
Appendix E. Quick Reference to vi Commands E-1
Appendix F. Summary of Shell Programming Ingredients F-1
Glossary G-1
Index I-1

HOW TO READ THIS GUIDE

The UNIX* system is a family of computer operating systems developed by AT&T Bell Laboratories and licensed by AT&T Technologies, Inc. Because it can run on many sizes and types of computers and because of all it can do, the UNIX system has gained wide popularity since it was introduced in the late 1960s. Now, either by choice or by fate, you are interested in learning something about it.

This guide is written to help you, the user, understand how the UNIX system works and what it can do for you. It introduces you to UNIX System V, Release 2. New versions of the UNIX system, called releases, will be offered as changes are made or as improvements are added.

Who Should Read This Guide

Whether you are a newcomer to the world of computers or an experienced computer user who is unfamiliar with the UNIX system, this guide is for you. Although it contains technical material, it can be understood by either a newcomer or an expert. You will find that learning to use the UNIX system requires some thought and time, but you will be rewarded with power and flexibility unattainable with other operating systems.

This guide assumes that you are one of a number of people using a computer on which the UNIX system is running, and that there is a person responsible for monitoring and controlling the UNIX system you are using. This person is the *system administrator*. If, however, you are using the UNIX system on a small computer, you may also act as its system administrator. In this case, in addition to this guide, you should consult the documents you received when the UNIX system programs were delivered to you. (See *Appendix A* for information on how to order additional copies.)

---

* Trademark of AT&T Bell Laboratories

How This Guide Is Organized

The material in this guide is organized into three major parts: *UNIX System Overview, UNIX System Tutorials,* and *Supplementary Information and Reference Tools.* Both the major parts and the chapters in each part are separated by tab dividers.

The following list summarizes the contents of each major part:

- *UNIX System Overview*--This part introduces you to the basic principles of the UNIX operating system. The material in this part is organized into four chapters, each chapter building on information presented in preceding chapters. Therefore, it is recommended that you read chapters 1 through 4 in order. The chapters that make up this part are:

  - *Chapter 1, What is the UNIX System?*--Acquaints you with the UNIX system and explains how it works.

  - *Chapter 2, Basics for UNIX System Users*--Covers topics related to using your terminal, obtaining a system account, and establishing contact with the UNIX system.

  - *Chapter 3, Using the File System*--Explains what the file system is, how you can organize information (data, text, and programs) using the file system, and how you can store and retrieve this information using appropriate commands.

  - *Chapter 4, UNIX System Capabilities*--Builds on material and terminology presented in the first three chapters. It highlights UNIX system capabilities, such as command execution, text editing, electronic communication, programming, and aids to software development.

- *UNIX System Tutorials*--Each chapter in this part takes a step-by-step approach to teach you about one aspect of the UNIX system. You will gain the greatest benefit from them if you work through the examples and exercises at a terminal connected to the UNIX system you will be using. The tutorials assume that

you understand the concepts introduced in chapters 1 through 4. For example, before reading either the *Line Editor Tutorial* or the *Screen Editor Tutorial,* read the explanation of text editors in *Chapter 4*. The chapters that make up this part are:

- *Chapter 5, Line Editor Tutorial*--Teaches you how to use the **ed** text editor to create and to modify text on a paper printing or a video display terminal.

- *Chapter 6, Screen Editor Tutorial*--Teaches you how to use the **vi*** text editor to create and to modify text on a video display terminal.

- *Chapter 7, Shell Tutorial*--Teaches you how to use the shell to automate repetitive jobs. The shell is the part of the UNIX system that interprets the commands you type.

- *Chapter 8, Communication Tutorial*--Teaches you how to send information to others, whether they are working on your UNIX system or on a different UNIX system.

• *Supplementary Information and Reference Tools*--This part is organized into six appendices, a glossary, and an index. This material contains additional information that you may find useful in learning about the UNIX system. The appendices are:

- *Appendix A, Selected UNIX System Documentation*--Lists additional UNIX system documentation that enhances or elaborates on the information presented in this guide. This appendix gives document titles, reference numbers, and information on how to obtain the documents.

- *Appendix B, File System Organization*--Illustrates how information is stored in the UNIX operating system.

- *Appendix C, Summary of UNIX System Commands*--Describes, in alphabetical order, each UNIX system command discussed in this guide.

---

* The visual editor is based on software developed by The University of California, Berkeley, California; Computer Service Division, Department of Electrical Engineering and Computer Science, and such software is owned and licensed by the Regents of the University of California.

- *Appendix D, Quick Reference to* **ed** *Commands*--Describes the commands used with the line editor (**ed**), first in alphabetical order, and then organized by topic, such as creating text, deleting text, and displaying text.

- *Appendix E, Quick Reference to* **vi** *Commands*--Describes the commands used with the screen editor (**vi**), first in alphabetical order, and then organized by topics, such as creating text, changing text, and cutting and pasting text.

- *Appendix F, Summary of Shell Programming Ingredients*--Describes shell command language concepts and shows how to use shell programming language statements.

Other sections in this part of the guide are:

- *Glossary*--Defines technical words and terms used in this book.

- *Index*--Gives an alphabetical listing of topics, together with the page numbers on which they appear in this guide.

Acknowledgements

Many persons, too numerous to mention, contributed suggestions that are reflected in the pages of this guide. These persons include members of AT&T Bell Laboratories and AT&T Technologies, Inc., as well as reviewers and consultants not affiliated with AT&T.

The text of this guide was prepared using UNIX system text editors described in this guide, formatted using the UNIX System *Documenter's Workbench** **troff**, **tbl**, and **mm** macros, and produced on an AUTOLOGIC, Inc., APS-5 phototypesetter operating under the UNIX system.

---

* Trademark of AT&T Technologies, Inc.

UNIX SYSTEM OVERVIEW

Contents

Chapter 1. What Is the UNIX System?

Chapter 2. Basics for UNIX System Users

Chapter 3. Using the File System

Chapter 4. UNIX System Capabilities

Chapter 1

WHAT IS THE UNIX SYSTEM?

PAGE

WHAT THE UNIX SYSTEM IS 1-1

HOW THE UNIX SYSTEM WORKS 1-3

- Kernel 1-4
- Shell 1-8
- Commands 1-9
  - What Commands Do 1-10
  - How Commands Execute 1-11

Chapter 1

WHAT IS THE UNIX SYSTEM?

WHAT THE UNIX SYSTEM IS

The UNIX system is a set of programs, called software, that acts as the link between a computer and you, its user. The UNIX system is designed to control the computer on which it is running so the computer can operate efficiently and smoothly and to provide you with an uncomplicated, efficient, and flexible computing environment.

UNIX system software does three things:

- It controls the computer,
- It acts as an interpreter between you and the computer, and
- It provides a package of programs or tools that allows you to do your work.

The UNIX system software that controls the computer is referred to as the operating system. The operating system coordinates all the details of the computer's internals, such as allocating system resources and making the computer available for general purposes. The nucleus of this operating system is called the kernel.

In the UNIX system, the software that acts as a liaison between you and the computer is called the shell. The shell interprets your requests and, if valid, retrieves programs from the computer's memory and executes them.

The UNIX system software that allows you to do your work includes programs and packages of programs called tools for electronic communication, for creating and changing text, and for writing programs and developing software tools.

Put simply, this package of services and utilities called the UNIX system offers:

- A *general purpose* system that makes the resources and capabilities of the computer available to you for performing a wide variety of jobs or applications, not simply one or a few specific tasks.

- A computing environment that allows for an *interactive* method of operation so you can directly communicate with the computer and receive an immediate response to your request or message.

- A technique for sharing what the system has to offer with other users, even though you have the impression that the UNIX system is giving you its undivided attention. This is called *timesharing*. The UNIX system creates this feeling by allowing you and other users--*multiusers*--slots of computing time measured in fractions of seconds. The rapidity and effectiveness with which the UNIX system switches from working with you to working with other users makes it appear that the system is working with all users simultaneously.

- A system that provides you with the capability of executing more than one program simultaneously, this feature is called *multitasking*.

The UNIX system, like other operating systems, gives the computer on which it runs a certain profile and distinguishing capabilities. But unlike other operating systems, it is largely machine-independent; this means that the UNIX system can run on mainframe computers as well as microcomputers and minicomputers.

From your point of view, regardless of the size or type of computer you are using, your computing environment will be the same. In fact, the integrity of the computing environment offered by the UNIX system remains intact, even with the addition of optional UNIX system software packages that enhance your computing capabilities.

HOW THE UNIX SYSTEM WORKS

After reading the past few pages, you know that the UNIX system offers you a set of software that performs services--some automatically, some you must request. You also know that the system creates a certain environment in which you can use its software. But before you can ask the UNIX system to do something, you need to know what it is capable of doing.

Look at *Figure 1-1*. It shows a set of layered circles in graduated sizes. Each circle represents specific UNIX system software, such as:

- Kernel,
- Shell, and
- Programs/tools that run on command.

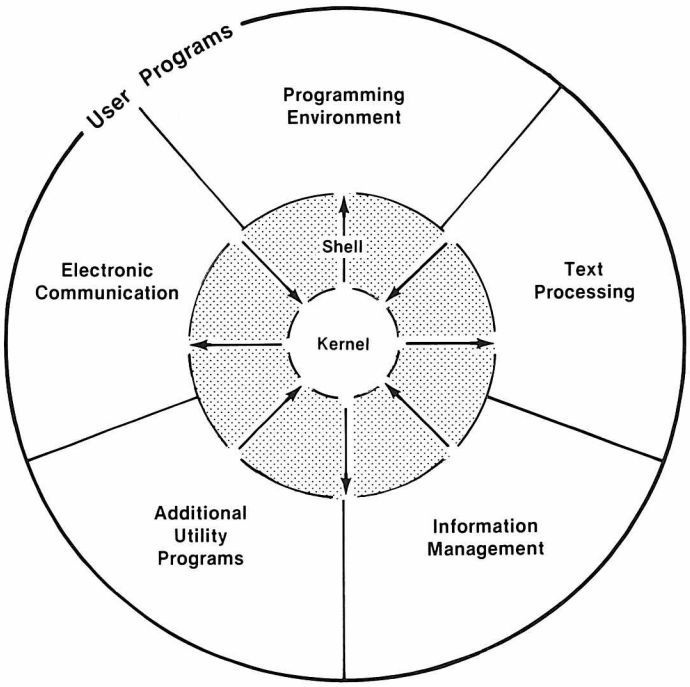


Figure 1-1. UNIX system model

You should know something about the major components of UNIX system software to communicate with the UNIX system. Therefore, the remainder of this chapter introduces you to each component: the kernel, the shell, and user programs or commands.

Kernel

The heart of the UNIX system is called the kernel. *Figure 1-2* gives an overview of the kernel's activities. Essentially, the kernel is software that controls access to the computer, manages the computer's memory, and allocates the computer's resources to one user, then to another. From your point of view, the kernel performs these tasks automatically. The details of how the kernel accomplishes this are hidden from you. This arrangement lets you focus on your work, not on the computer's.

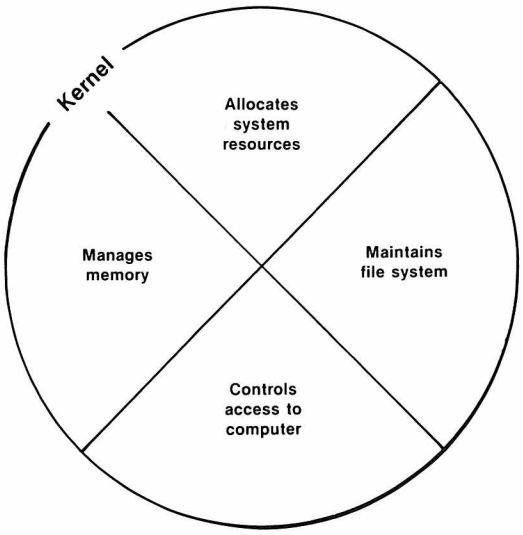


Figure 1-2. Functional view of kernel

On the other hand, you will become increasingly familiar with another feature of the kernel; this feature is referred to as the file system.

The file system is the cornerstone of the UNIX operating system. It provides you with a logical, straightforward way to organize, retrieve, and manage information electronically. If it were possible to see this file system, it might look like an inverted tree or organization chart made up of various types of files *Figure 1-3*. The file is the basic unit of the UNIX system and it can be any one of three types:

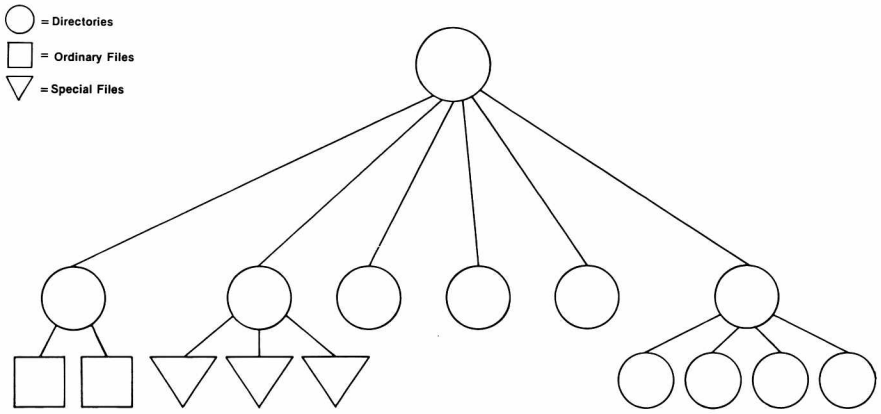


Figure 1-3. Branching directories and files give the UNIX system its treelike structure

- An *ordinary file* is simply a collection of characters. Ordinary files are used to store information. They may contain text or data for the letters or reports you type, code for the programs you write, or commands to run your programs. In the UNIX system, everything you wish to save must be written into a file.

  In other words, a file is a place for you to put information for safekeeping until you need to recall or use its contents again. You can add material to or delete material from a file once you have created it, or you can remove it entirely when the file is no longer needed.

- A *directory* is a file maintained by the operating system for organizing the treelike structure of the file system. A directory contains files and other directories as designated by you. You can build a directory to hold or organize your files on the basis of some similarity or criterion, such as subject or type.

  For example, a directory might hold files containing memos and reports you write pertaining to a specific project or client. Or a directory might hold files containing research specifications and programming source code for product development. A directory might hold files of executable code allowing you to run your computing jobs. Or a directory might contain files representing any combination of these possibilities.

- A *special file* represents a physical device, such as the terminal on which you do your computing work or a disk on which ordinary files are stored. At least one special file corresponds to each physical device supported by the UNIX system.

In some operating systems, you must define the kind of file you will be working with and then use it in a specified way. You must consider how the files are stored since they can be sequential, random-access, or binary files. To the UNIX system, however, all files are alike. This makes the UNIX system file structure easy to use. For example, you need not specify memory requirements for your files since the system automatically does this for you. Or if you or a program you write needs to access a certain device, such as a printer, you specify the device just as you would another one of your files. In the UNIX system, there is only one interface for all input from you and output to you; this simplifies your interaction with the system.

The source of the UNIX system file structure is a directory known as root, which is designated with a slash (/). All files and directories in the file system are arranged in a hierarchy under root. Root normally contains the kernel as well as links to several important system directories that are shown in *Figure 1-4*:

/bin Many executable programs and utilities reside in this directory.

/dev This directory contains special files that represent peripheral devices, such as the console, the line printer, user terminals, and disks.

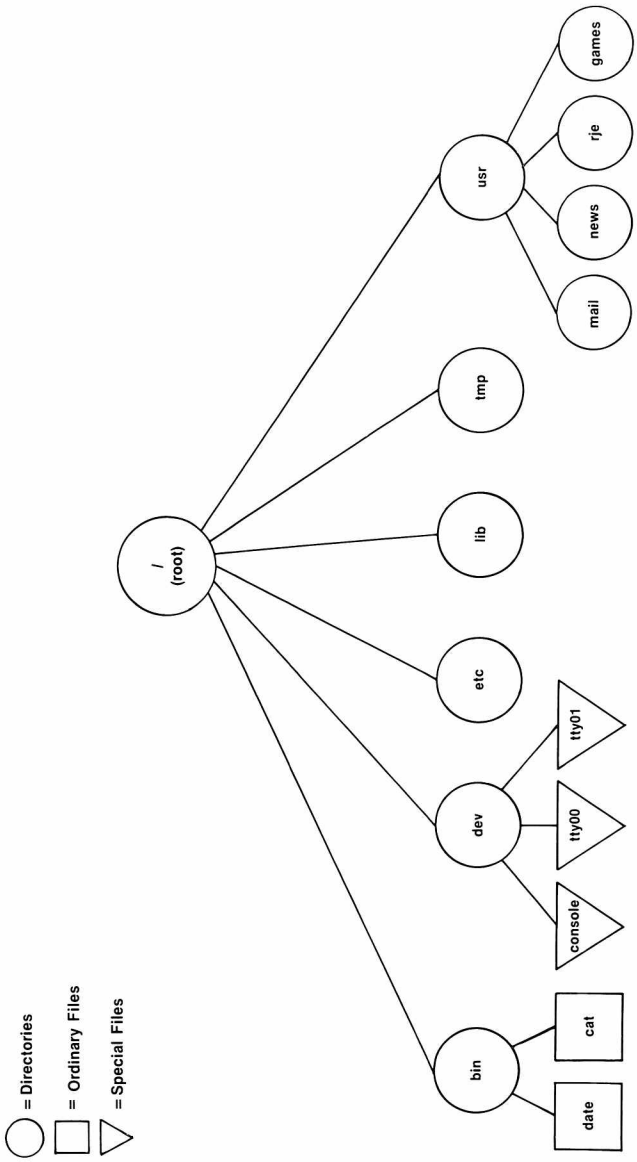


Figure 1-4. Sample of typical file system structure

/etc Programs and data files for system administration can be found in this directory.

/lib This directory contains available program and language libraries.

/tmp This directory is a place where anyone can create temporary files.

/usr This directory holds other directories, such as **mail** (which further holds files storing electronic mail), **news** (which contains files holding newsworthy items), **rje** (which contains files needed to send data via something called the **r**emote **j**ob **e**ntry communication link), and **games** (which contains files holding electronic games).

In summary, the directories and files you create comprise the portion of the file system that is structured and, for the most part, controlled by you. Other parts of the file system are provided and maintained by the operating system, such as **bin, dev, etc, lib, tmp** and **usr**, and have much the same structure on all UNIX systems.

Chapter 3 shows how to organize a file system directory structure and how to access and manipulate files. *Chapter 4* gives an overview of UNIX system capabilities. The effective use of these capabilities depends on your familiarity with the file system and your ability to access information stored within it. *Chapter 5* and *Chapter 6* are tutorials designed to teach you how to create and edit files to meet your computing and information management needs.

Shell

The shell is a unique UNIX system program or tool that is central to most of your interactions with the UNIX system. *Figure 1-1* illustrates how the shell works. The drawing shows the shell as a circle containing arrows pointing away from the kernel and the file system to the outer circle that contains programs and then back again. The arrows indicate that a two-way flow of communication is possible between you and the computer via the shell.

When you enter a request to the UNIX system by typing on the terminal keyboard, the shell translates your request into language the computer understands. If your request is valid, the computer honors it and carries out an instruction or set of instructions. Because of its job as translator, the shell is called the command language interpreter.

As the command language interpreter, the shell can also help you to manage information. The shell's ability to manage information stems from the design of the UNIX system. Each program in the UNIX system is designed to do one thing well. In a sense, a UNIX system program is a building block or module that you can use in tandem with other programs to create even more powerful tools.

In addition to acting as a command language interpreter, the shell is a programming language complete with variables and control flow capabilities.

A section of *Chapter 4* describes each of the shell's capabilities. *Chapter 7* teaches you how to use these capabilities to write simple shell programs called shell scripts and how to custom-tailor your computing environment.

Commands

A program is a set of instructions that the computer follows to do a specific job. In the UNIX system, programs that can be executed by the computer without need for translation are called executable programs or commands.

As a typical user of the UNIX system, you have many standard programs and tools available to you. If you also use the UNIX system to write programs and to design and develop software, you have system calls, subroutines, and other tools at your disposal. And you have, of course, the programs you write.

This book introduces you to approximately 40 of the most frequently used programs and tools that you will probably use on a regular basis when you interact with the UNIX system. If you need additional information on these or other standard UNIX system programs, check the *UNIX System User Reference Manual.* If you want to use tools and

routines that relate to programming and software development, you should consult the *UNIX System Programmer Reference Manual* and the *UNIX System Support Tools Guide*. *Appendix A* provides you with information on how to obtain copies of these manuals.

The details contained in the two reference manuals may also be available via your terminal in what is called the *on-line* version of the UNIX system reference manuals. This on-line version is made up of formatted text files that look exactly like the printed pages in the manuals. You can summon pages in this electronic manual using the command **man**, which stands for **man**ual page. If the electronic version of the manuals is available on your computer, the **man** command is documented in your copy of the *UNIX System User Reference Manual*.

What Commands Do

The outer circle of *Figure 1-1* organizes UNIX system programs and tools into general categories according to what they do. The programs and tools allow you to:

- *Process text*. This capability includes programs, such as, line and screen editors (which create and change text), a spelling checker (which locates spelling errors), and optional text formatters (which produce high-quality paper copies that are suitable for publication).

- *Manage information*. The UNIX system provides many programs that allow you to create, organize, and remove files and directories.

- *Communicate electronically*. Several programs, such as **mail**, provide you with the capability to transmit information to other users and to other UNIX systems.

- *Use a productive programming and software development environment*. A number of UNIX system programs establish a friendly programming environment by providing UNIX-to-programming-language interfaces and by supplying numerous utility programs.

- *Take advantage of additional system capabilities*. These programs include graphics, a desk calculator package, and computer games.

How Commands Execute

Figure 1-5 gives a general idea of what happens when the UNIX system executes a command.

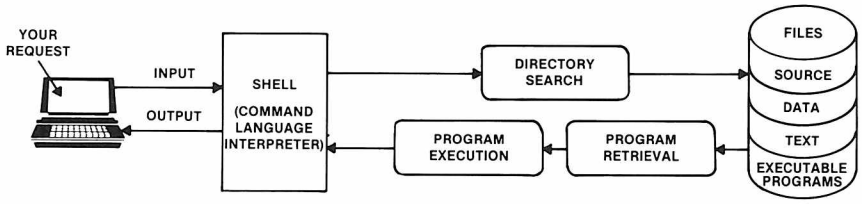


Figure 1-5. Flow of control between you and computer when you request program to run

When the shell signals it is ready to accept your request, you type in the command you wish to execute on the keyboard. The command is considered input, and the shell searches one or more directories to locate the program you specified. When the program is found, the shell brings your request to the attention of the kernel. The kernel then follows the program's instructions and executes your request. After the program runs, the shell asks you for more information or tells you it is ready for your next command.

This is how the UNIX system works when your request is in a format that the shell understands. The structure that the shell understands is called a command line. *Chapter 3* explains what you need to know about the command line so you can request a program to run.

This chapter has outlined some basic principles of the UNIX operating system and explained how they work. The following chapters will help you begin to apply these principles according to your computing needs.

Chapter 2

BASICS FOR UNIX SYSTEM USERS

PAGE

GETTING STARTED 2-1

ABOUT THE TERMINAL 2-2
- Required Terminal Settings 2-3
- Keyboard Characteristics 2-4
- Typing Conventions 2-6
  - Responding to the Command Prompt 2-8
  - Correcting Typing Errors 2-8
  - Typing Speed 2-9
  - Stopping a Command 2-9
  - Using Control Charcters 2-9

OBTAINING A LOGIN NAME 2-11

ESTABLISHING CONTACT WITH THE UNIX SYSTEM 2-11
- Login Procedure 2-13
- Password 2-14
- External Security Code 2-16
- Possible Problems When Logging In 2-17
- Simple Commands 2-19
- Logging Off 2-20

Chapter 2

BASICS FOR UNIX SYSTEM USERS

GETTING STARTED

There are general rules and guidelines with which you should be familiar before you begin to work on the UNIX system. For example, you need information about your terminal and how to use its keyboard and about how to begin and end a computing session.

This chapter acquaints you with these rules and guidelines and presents you with information to help to make your first encounter with the UNIX system understandable and to lay the groundwork for future computing sessions. Since the best way to learn about the UNIX system is to use it, this chapter helps to get you started by providing examples of how to use these rules and guidelines to establish contact with the UNIX system and to respond to its requests and prompts.

For your convenience, an outline of a terminal display screen is used to set off examples of interactions between you and the UNIX system. These examples apply regardless of the type of terminal you use. Inside the screen, what the UNIX system prompts and its responses are printed in *italic*. The commands you type in response to the system prompts and your other input and data are printed in **bold**face type. These include the commands you type that do not appear on the screen (such as, a carriage return), which are enclosed in angle brackets < >. The following screen summarizes these conventions.

italic	(UNIX system prompts and responses)
bold	(Your commands)
<>	(Your commands or parts of commands that do not appear on the screen)

Without further ado, let's begin.

To establish contact with the UNIX system, you need:

- A terminal,
- An identification name, called a login name, by which the UNIX system recognizes you as one of its authorized users,
- A password with which the UNIX system double-checks and verifies your identity after you log in and before it allows you to use its resources, and
- The telephone number to the UNIX system to which your login name is assigned if your terminal is not directly connected or wired to the computer.

ABOUT THE TERMINAL

A terminal is an input/output device: through it you input a request to the UNIX system and the system, in turn, outputs a response to you. The terminal is equipped with a keyboard, a monitor or display unit (much like the screen on a television set), a control unit, and a link that allows it to communicate with the computer.

The terminal you use to interact with the UNIX system can be either a video display terminal or a printing terminal (*Figure 2-1*).

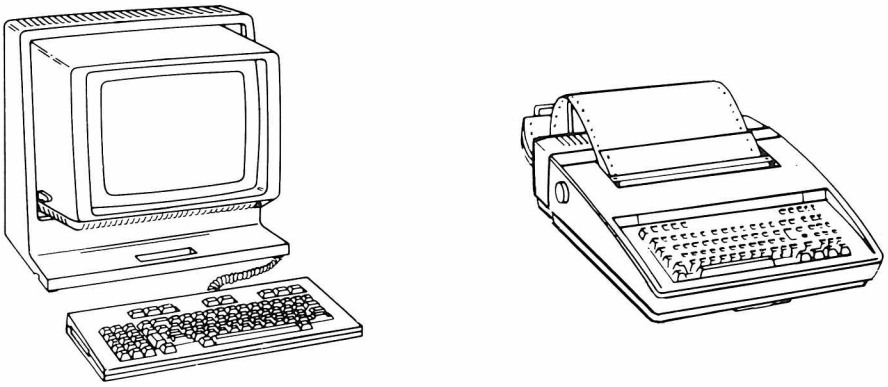


Figure 2-1. Left, video display terminal (TELETYPE® 5410); right, printing terminal (TELETYPE 43)

These terminals differ in how they monitor or display input/output. The video display terminal uses a display screen, whereas the printing terminal uses continuously fed paper.

Required Terminal Settings

Regardless of the type of terminal you use, you must set it up or configure it in a certain way to insure proper communication with the UNIX system.

If you have not set terminal options before, you might feel more comfortable seeking help from someone who has. Or you can, of course, be adventurous.

® Registered trademark of Teletype Corporation

How you configure a terminal depends on the type of terminal that you are using. Some terminals are configured with switches, whereas other terminals are configured directly from the keyboard using a set of function keys. To determine how to configure your terminal, consult the owner's manual provided by the manufacturer.

Following is a list of configuration checks to be performed on any terminal before attempting to establish contact with the UNIX system.

- Turn on the power.
- Set the terminal to ON-LINE or REMOTE operation. This setting insures that the terminal is under direct control of the computer.
- Set the terminal to FULL DUPLEX mode. The full duplex mode insures two-way communication or input/output between you and the UNIX system.
- If your terminal is not directly connected or hard wired to the computer, make sure the acoustic coupler or data phone set you are using is set to the FULL DUPLEX mode.
- Set character generation to LOWERCASE. If the terminal, however, generates only uppercase letters, the UNIX system will accommodate it by printing everything that transpires during the computing session in uppercase letters.
- Set the terminal to NO PARITY.
- Set the speed or rate at which the computer communicates with the terminal. This rate of communication is called the baud rate. Typical terminal speeds are 30 and 120 characters per second or 300 and 1,200 baud, respectively. Occasionally, speeds such as 240, 480, and 960 characters per second or 2,400, 4,800, and 9,600 baud, respectively, are available.

Keyboard Characteristics

If you have seen or had some experience with a typewriter, the keyboard shown in *Figure 2-2* should look somewhat familiar.

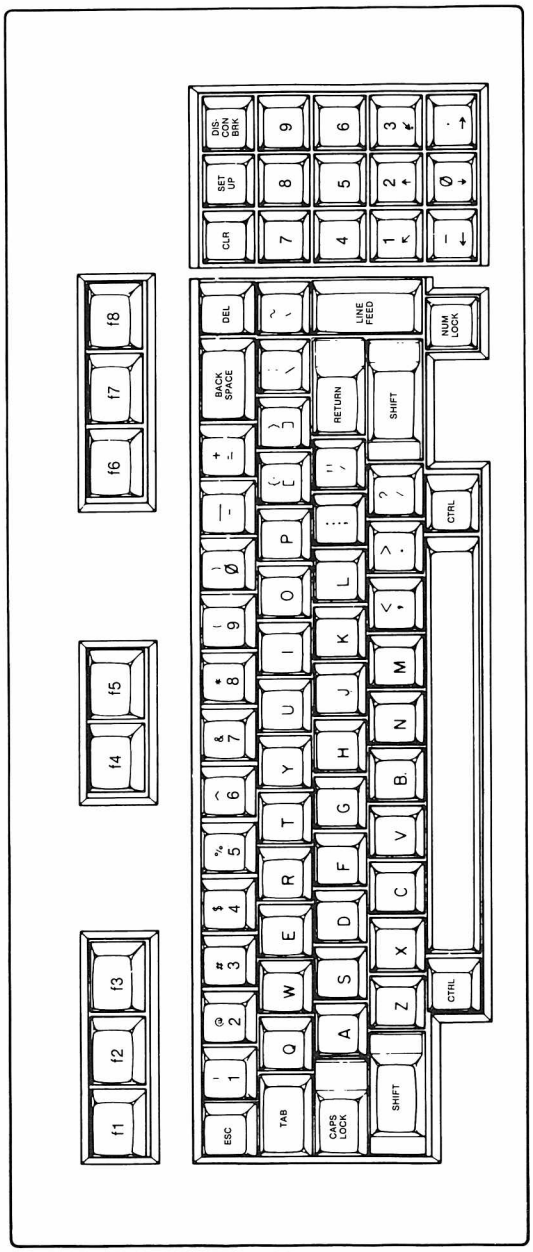


Figure 2-2. Example of keyboard layout (TELETYPE 5410)

Its keys correspond to:

- Letters of the English alphabet **a** through **z** and **A** through **Z** when you are holding down a shift key,
- Numeric characters 0 through 9,
- A variety of symbols, such as ! @ # $ % ^ & ( ) _ − + = ~ ' { } [ ] \ : ; " ' < > , ? /
- Words, such as RETURN and BREAK, and abbreviations, such as DEL (delete), CTRL (control), and ESC (escape).

Many of the keys corresponding to symbols, words, and abbreviations have been added to the keyboard layout and the placement of these characters or symbols on a keyboard may vary from terminal to terminal.

Consequently, there is not a truly standard layout for terminal keyboard characters. There is, however, a standard set of characters that keyboards have, consisting of 128 characters, called the ASCII character set. ASCII is pronounced "*as kee*" and is the abbreviation for American Standard Code for Information Interchange. When you depress a key or combination of keys, the appropriate ASCII code is sent to the computer for translation from the alphabetic and numeric characters that we understand to electronic signals that the computer can decode.

Typing Conventions

To interact effectively with the UNIX system, you should be familiar with certain typing conventions. An example of a UNIX system typing convention is using lowercase letters when you issue commands. Other typing conventions require that you use specific characters to erase letters and delete lines, or combinations of characters to stop the UNIX system from printing output on your terminal monitor temporarily.

The next few pages introduce you to these conventions. *Table 2-1* lists these special characters, keystrokes, and their meanings for your quick reference.

TABLE 2-1
UNIX System Typing Conventions

Key(s)	Meaning
$	System's command prompt (your cue to respond)
#	Erase a character
@	Erase or kill an entire line
BREAK*	Stop execution of a program or command
DEL*	Delete or kill the current command line
ESC*	Use with another character to perform specific function (called escape sequence)
	OR
	Use to indicate end of create mode when using screen editor (**vi**)
RETURN*	End a line of typing; designated as **<CR>**
Control d*	Stop input to system or log off; designated as **<^d>**
Control h*	Backspace for terminals without a backspace key; designated as **<^h>**
Control i*	Horizontal tab for terminals without a tab key; designated as **<^i>**
Control s*	Temporarily stops output from printing on screen; designated as **<^s>**
Control q*	Resumes printing after typing **<^s>**; designated as **<^q>**

NOTE: All control characters are sent by holding down the control key and pressing the appropriate letter.

* Nonprinting characters.

Responding to the Command Prompt

The standard UNIX system command prompt is the dollar sign, **$**. When the **$** appears on your terminal monitor, it means that the UNIX system is waiting for you to tell it to do something. Your response to the **$** prompt is to issue commands followed by depressing the carriage return key, designated as **<CR>** throughout this guide.

The **$** is the default value for the command prompt. *Chapter 7* explains how to change the default value to another prompt.

Correcting Typing Errors

You can correct typing errors in two ways providing you have not pressed **<CR>**. The # symbol allows you to erase previously typed characters on a line, and the @ sign allows you to delete the line on which you are working. The # and the @ characters are default values for character and line deletion, respectively.

Pressing the # key erases the character previously typed, whereas repetitive use of the # sign erases any number of characters back to the beginning of the line, but not beyond that. For example, typing

helo#lo

on your terminal keyboard is interpreted by the UNIX system as "hello" correctly typed.

To delete the entire line on which you are working, press the @ key. When you do, the UNIX system moves you to the beginning of the next line.

If you want to use the # or the @ characters literally, that is, you would like a file to contain the line

Only one # appears on this sheet of music.

or

I purchased three books @ $15.75 per book.

you would have to press the backslash (\) key before pressing the # key. Otherwise, the # would erase the space after the word "one" and the line would print as

Only one appears on this sheet of music.

If you press the @ key without first pressing the \ key while typing the second example, the @ would erase the entire line. On the other hand, the leading \ removes the special meaning attached to characters like # and @ so that they can be understood literally by the computer.

Typing Speed

After the $ appears on your terminal monitor you can type as fast as you want, even during periods when the UNIX system is responding to or executing a command. The printout on your terminal monitor will appear garbled because your input is intermixed with the system's output. The UNIX system, however, has what is referred to as read-ahead capability, which allows it to separate input from output and to respond to your command properly.

With read-ahead capability, the UNIX system stores your next request while the system is outputting information on your terminal monitor in response to a previous request.

Stopping a Command

If you wish to stop the execution of a command, simply depress the BREAK or DEL key. In turn, you will receive the $ prompt indicating that the UNIX system terminated the running of the program and is ready to accept your next command.

Using Control Characters

Locate the control key on your terminal keyboard. The key may be labeled CTRL or CONTROL and is probably to the left of the A key or below the Z key. The control character is used in combination with other keyboard characters to initiate a physical controlling action across a line of typing, such as backspacing or tabbing. In addition, some control characters define UNIX-system-specific

commands, such as temporarily halting output from printing on a terminal monitor.

Type a control character by holding down the CTRL key and depressing an appropriate alphabetic key. Control characters do not print on the terminal when typed. In this book, control characters are designated with a preceding carat (^), such as **<^s>** for control s, to help identify them.

Let's take a look at the capabilities of the control character combinations you will be using regularly when working with the UNIX system.

Temporarily Stopping Output. At times, you may wish to stop the UNIX system temporarily from printing output on your terminal monitor. This could surely be the case when you wish to keep information from rolling off the screen monitor on a video display terminal. If you type **<^s>**, printing of output ceases; typing **<^q>** causes the printing to resume.

Terminating a Computing Session. When you have completed a session with the UNIX system, you should type **<^d>**. This is the recommended way to log off the system and is described in detail later in this chapter.

Additional Control Character Capabilities. The UNIX system furnishes other control character capabilities. For instance, if your terminal keyboard does not have a backspace key, typing **<^h>** gives you a backspace. Typing **<^i>** gives you a tab key if your terminal is set properly. (Refer to the section entitled *Possible Problems When Logging In* for information on how to set the tab key.)

After you configure the terminal and survey its keyboard, you are ready to establish communication with the UNIX system if you have a login name.

OBTAINING A LOGIN NAME

Generally speaking, a log contains a record of information or data that notes a series of events or measures progress or performance.

The UNIX system procedure for logging in is based on this idea. When you attempt to establish contact with the system, the UNIX system verifies that you are an authorized user. If you pass the system's security checks, the UNIX system allows you to log in. After you are logged in, the system maintains a record of the resources you use, the way in which you use them, and for how long. This log helps the people who manage and maintain the system by giving them complete user and resource allocation information.

To receive a login name, set up a UNIX system account through your local system administrator or the person in charge of your UNIX system installation. When the account is approved you should receive notification of your login name and the telephone number of the system to which your login is assigned.

Your login name is determined by local practices. Possible examples are your last name, your nickname, or a UNIX system account number. Typically, a login name is three to eight characters in length. It can contain any combination of alphanumeric characters, as long as it starts with a letter. It cannot, however, contain any symbols. According to these rules, the following examples are legal UNIX system login names: *starship*, *mary2*, and *jmrs*.

ESTABLISHING CONTACT WITH THE UNIX SYSTEM

When you attempt to contact the UNIX system, you will typically be using a terminal that is directly wired to a computer or a terminal that communicates with the system via a telephone connection.

If your terminal has a direct-wired connection, turn on the power and the message *login* should appear on the upper left side of the screen monitor or paper display.

If you must establish a dial-in connection, do the following:

1. Dial the telephone number that connects you to the UNIX system. You will hear one of the following:

   - Busy signal, which means circuits are busy. Hang up and dial again.

   - Continuous ringing and no answer. This usually indicates that there is trouble with the telephone line or that the system is inoperable because of mechanical failure or electronic problems. Hang up and dial again later.

   - A high-pitched tone, which indicates the system is accessible.

2. When you receive the high-pitched tone, place the handset of the phone in the acoustic coupler or momentarily depress the appropriate button on the data phone set (you can determine this by referring to the owner's manual for the equipment) and then replace the handset in the cradle (*Figure 2-3*).

3. After a few seconds, the UNIX system should display the *login* prompt.

4. If you are greeted with a series of meaningless characters, the telephone number you called serves more than one baud rate and the UNIX system is trying to communicate with you but is using the wrong speed. Depress the BREAK or RETURN key, which signals the system to try another speed. If the UNIX system does not display the *login* prompt within a few seconds, depress the BREAK or RETURN key once again.

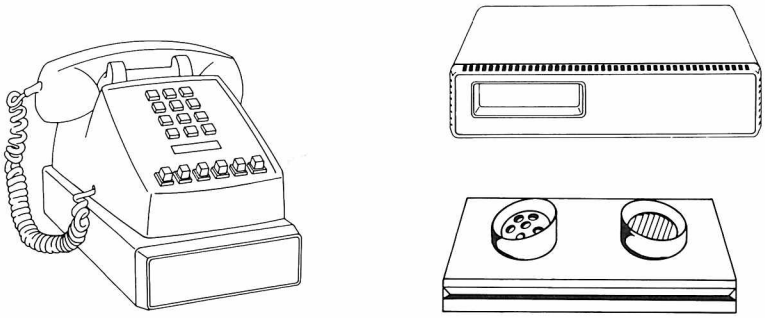


Figure 2-3. Left, data phone set (Data Set 212A*); right, modem for data phone set (DATAPHONE® II Modem); lower right, acoustic coupler

Login Procedure

When the connection is made and the UNIX system prompts for your login name, type in your login name and depress **<CR>**. In the following examples, *starship* is the login name.

login: **starship<CR>**

Remember to type in lowercase letters. If you use uppercase letters, the UNIX system will also use uppercase letters until you log out and log in again.

---

* Manufactured by AT&T Technologies, Inc.

® Registered trademark of AT&T

Password

After typing in your login name, the UNIX system prompts you for your password. In a typical session, you would simply type in your password followed by **<CR>**. For security reasons, the UNIX system will not print (echo) your password on the terminal monitor.

If both your login name and password are acceptable to the UNIX system, the system prints newsworthy messages for users. These items might include details about a new system tool or furnish a schedule for system maintenance. The news items are followed by the UNIX system command prompt, which is the **$** symbol.

Your terminal monitor should look something like the one that follows when you complete the login sequence successfully:

```
login: starship<CR>
password:
UNIX system news
$
```

If you made a typing mistake that you did not correct before depressing **<CR>**, the UNIX system displays the message *login incorrect* on your terminal monitor and asks you to try again by printing the login prompt. It is also possible that your communication link with the UNIX system might be dropped in which case you would have to try to log in again.

```
login: ttarship<CR>
password:
login incorrect
login:
```

If you have never logged into the UNIX system, your login procedure will differ somewhat from the typical one just described. This is because as a first-time user you were probably assigned a temporary password when your system account was set up and the system will not allow you to access its resources until you choose a new one.

This extra step maintains a security requirement, which is that you choose a password for your exclusive use. Protection of system resources and your personal files depends on you keeping the password you select private.

The actual procedure you will follow is determined according to administration procedures at your computer installation site. A typical example of what you might be expected to do if you have a new UNIX system account and you are logging in for the first time follows.

1. The UNIX system displays the login prompt when you establish contact with it. You should type in your login name followed by **<CR>.**

2. When the UNIX system prints the password prompt, you should type in your temporary password and depress **<CR>.**

3. At this point, the system tells you the temporary password has expired and that it is time to select a new one.

4. The UNIX system asks you to input the old password again. Type in your temporary password.

5. The system prompts you to input your new password. Type in the password you choose.

   The password you select is usually six to eight characters in length and contains at least one numeric character. In addition, you can also use special characters. Examples of valid passwords are: *mar84ch, JonathOn,* and *BRAV3S.*

   The UNIX system you are using may have different requirements to consider when choosing a password. Ask another system user or contact the system administrator if you are not sure of the specifics.

6. For verification, the system requests that you re-enter your new password. Type in the new password once again.

   This is a valuable check for you and the UNIX system since a password is not printed on the terminal monitor.

7. If you do not re-enter the new password exactly as you typed it the first time, the system tells you that the passwords do not match and asks you to try the procedure again. On some systems, however, the communication link may be dropped if you do not re-enter the password exactly as you typed it the first time. If this is the case, you must begin the login procedure again.

   When the passwords match, the system displays the **$** command prompt.

The following screen summarizes this procedure for first-time UNIX system users.

```
login: starship                    <CR>
password:                          <CR>
Your password has expired.
Choose a new one.
Old password:                      <CR>
New password:                      <CR>
Re-enter new password:             <CR>
UNIX system news
$
```

External Security Code

If you are able to access the UNIX system from outside your computer installation site, you may need additional information to establish contact with the UNIX system, such as a special telephone number or another security code. To determine if this feature is available to you, contact your system administrator.

Possible Problems When Logging In

A terminal usually behaves predictably providing you have configured it properly. Sometimes, however, it may act peculiarly. For example, each character you type may appear twice on the terminal monitor or the carriage return may not work properly.

Some problems can be corrected by simply logging off the system and logging on again. If logging on a second time does not remedy the problem, you should first check the following and try logging in once again:

- *Keyboard*--Keys that are marked CAPS, LOCAL, BLOCK, and so on should not be enabled, that is, in the locked position. You can usually disable these keys simply by depressing them.

- *Data phone set or modem*--If your terminal is connected to the computer via telephone lines, verify that the baud rate and duplex settings are correctly specified.

- *Switches*--Some terminals have several switches that must be set to be compatible with the UNIX system. If this is the case with the terminal you are using, make sure they are set properly.

Refer to the section *Required Terminal Settings* in this chapter if you need information to verify the terminal configuration. If you need additional information about the keyboard, terminal, and data phone or modem, check the owner's manuals for the equipment.

Table 2-2 presents a list of procedures you can follow to detect, diagnose, and correct some problems you may experience when trying to establish contact with the UNIX system. If none of the possibilities covered in the table helps you, contact the system administrator or the person in charge of the UNIX installation at your location.

TABLE 2-2
Troubleshooting Problems When Logging in*

Problem†	Possible Cause	Action/Remedy
Stream of meaningless characters when logging in	UNIX system attempting to communicate at wrong speed	Depress RETURN or BREAK key
Input and output is printed in uppercase letters	Terminal configuration includes UPPERCASE setting	Log off, set character generation to LOWERCASE, and log in again
Input is printed in UPPERCASE letters, output in LOWERCASE	Key marked CAPS or CAPS LOCK is locked or enabled	Depress the CAPS or CAPS LOCK key to disable setting
Input is printed (echoed) twice	Terminal is set to HALF DUPLEX mode	Change setting to FULL DUPLEX mode
Tab key does not work properly	Tabs are not set to advance to next	Type **stty -tabs**‡
Communication link cannot be established in spite of receiving high pitched tone when dialing in	Terminal is set to LOCAL or OFF-LINE mode	Set terminal to ON-LINE operation and try logging in again
Communication link between terminal and UNIX system is repeatedly dropped on logging in	Terminal is set to LOCAL or OFF-LINE mode	Call system administrator

* Numerous problems can occur if your terminal is not configured properly. To eliminate these possibilities before attempting to log in, perform the configuration checks listed on page 2-4.

† Some problems may be specific to your terminal, data set, or modem, check the owner's manual for this equipment if suggested actions do not remedy the problem.

‡ Typing **stty -tabs** corrects tab setting only for your current computing session. To insure correct tab setting for all sessions, add the line **stty -tabs** to your profile (see *Chapter 7*).

Simple Commands

When the **$** command prompt is displayed on your monitor, you know that the UNIX system recognizes you as an authorized user. Your response to the **$** command prompt is to request UNIX system programs to run.

Type in the command **date** and press **<CR>** after the command prompt. When you do this, the UNIX system retrieves the **date** program and executes it. As a result, your terminal monitor should look something like the following.

```
$ date<CR>
Wed Oct 12 09:49:44 CDT 1983
$
```

As you can see, the UNIX system prints the date and the time. In this example, the CDT stands for Central Daylight Time. Your terminal monitor will display the appropriate time for your geographical location.

Now type the command **who** and depress **<CR>**. Your screen will look something like this.

```
$ who<CR>
starship     tty00     Oct 12  8:53
mary2        tty02     Oct 12  8:56
acct123      tty05     Oct 12  8:54
jmrs         tty06     Oct 12  8:56
$
```

The **who** command lists the login names of everyone currently working on your system. The tty designations refer to the names of the special files that correspond to the terminals on which you and other users are currently working. The login date and time for each are also given.

Logging Off

When you have completed a session with the UNIX system, you should type **<^d>** after the **$** command prompt. (Remember that control characters such as the **<^d>** are typed by holding down the control key and depressing the appropriate alphabetic key.) Since they are nonprinting characters, they do not appear on the terminal monitor. In a few seconds, the UNIX system should display the login message again. This indicates you have logged off successfully and someone else can log in at this time. Your terminal monitor should look like the one that follows.

```
$ <^d>
login:
```

It is strongly recommended that you log off the system using **<^d>** before turning off the terminal or hanging up the phone. It is the only way to assure you have been logged off the UNIX system.

Chapter 3

USING THE FILE SYSTEM

PAGE

INTRODUCTION 3-1

HOW THE FILE SYSTEM IS STRUCTURED 3-4

YOUR PLACE IN THE FILE SYSTEM STRUCTURE 3-4
- Your Home Directory 3-6
- Your Working Directory 3-6
- Path Names 3-9
  - Full Path Names 3-9
  - Relative Path Names 3-12

ORGANIZING A DIRECTORY STRUCTURE 3-16
- Creating Directories *(mkdir)* 3-16
- Listing the Contents of a Directory *(ls)* 3-19
  - Frequently Used *ls* Options 3-21
  - Command Summary 3-24
- Changing Your Working Directory *(cd)* 3-25
- Removing Directories *(rmdir)* 3-27

ACCESSING AND MANIPULATING FILES 3-29
- Basic Commands 3-29
  - Displaying a File's Contents *(cat, pg, pr)* 3-30
  - Requesting a Paper Copy of a File *(lp)* 3-39
  - Making a Duplicate Copy of a File *(cp)* 3-41
  - Moving and Renaming a File *(mv)* 3-44
  - Removing a File *(rm)* 3-46
  - Counting Lines, Words, and Characters in a File *(wc)* 3-47
  - Protecting Your Files *(chmod)* 3-51
- Advanced Commands 3-56
  - Identifying Differences Between Files *(diff)* 3-57
  - Searching a File for a Pattern *(grep)* 3-59
  - Sorting and Merging Files *(sort)* 3-62

SUMMARY 3-64

Chapter 3

USING THE FILE SYSTEM

INTRODUCTION

To use the file system effectively you must be familiar with its structure, know something about your relationship to this structure, and understand how the relationship changes as you move around within it. Reading this chapter serves as preparation to use this file system.

The first ten or so pages should help to give you a working perspective of the file system. These pages contain information on the makeup of the file system and on how you fit into its organization. The remainder of the chapter introduces you to a number of UNIX system commands. Some you can use to build your own directory structure, whereas others allow you to access and manipulate the subdirectories and files you organize within it. And others still allow you to examine the contents of other directories in the system that you have permission to look at or to use.

Each command is discussed in a separate subsection in a way that will allow you to use it effectively. Many of the commands presented in this section have additional, sophisticated uses; these, however, are left for more experienced users and are described in other UNIX system documentation. You can choose to read these sections in the order in which they are presented in the text or you can opt to read about the commands and their capabilities in the order that best suits your interests and purpose. Nevertheless, all the commands presented are basic to using the file system efficiently and easily. It is recommended that you read through them thoroughly and then try them out. Before viewing how the file system is structured, however, let's take a look at the structure of a command.

For the UNIX system to understand your intentions when using commands, you must take care to see that you input commands using the correct format, called the command line syntax. The command line syntax provides a procedure for ordering elements in a command line. It serves the same purpose as putting words in a certain sequence or order so that you can meaningfully express your ideas and thoughts to others. Without sentence structure, people would have difficulty interpreting what you mean. Similarly, without command line syntax, the UNIX system shell cannot interpret your request.

Command line syntax consists of one or more of the following elements separated by a blank or blanks and followed by pressing the carriage return **<CR>** key:

command option(s) argument(s)

where

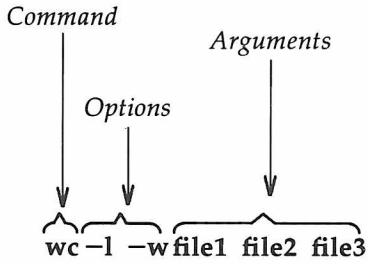
command is the name of the program you wish to run,

option modifies how the command runs, and

argument specifies data on which the command is to focus or operate (usually a directory or file name).

A command line can simply contain a command name followed by **<CR>**, or it can list options and/or arguments in addition to the command. If you specify options and arguments on the command line, you must separate them with at least one blank. Blanks can be typed by pressing the space bar or the tab key. If a blank is part of the argument name, enclose the argument in double quotation marks, for example, "sample 1".

Some commands allow you to specify multiple options and/or arguments on a command line. Consider the following command line:



In this example, **wc** is the name of the command and two options **−l** and **−w** have been specified. (The UNIX system usually allows you to group options such as these to read **−lw** if you prefer.) In addition, three files--*file1, file2,* and *file3*--are specified as arguments. Although most options can be grouped together, arguments cannot.

The following examples show the proper sequence and spacing in command line syntax:

Incorrect	**Correct**
wcfile	wc file
wc−lfile	wc −l file
wc −l w file	wc −lw file
	or
	wc −l −w file
wc file1file2	wc file1 file2

You can refer back to the ground rules on command line syntax as you read and work through the chapter.

HOW THE FILE SYSTEM IS STRUCTURED

The file system is comprised of a set of directories, ordinary files, and special files. These components provide you with a way to organize, retrieve, and manage information electronically. *Chapter 1* introduced you to directories and files, but let's review what they are before learning how to use them to tap the resources of the file system.

In general, a directory is a collection of files and other directories. Specifically, it contains the names of these files and directories. You can build a directory to organize the files you create on the basis of some similarity. An ordinary file is a collection of characters that is stored on a disk. Such a file may contain text for a status report you type or code for a program you write. Any information you wish to save must be written into a file. And a special file represents a physical device, such as your terminal.

The set of all the directories and files is organized into a treelike structure. *Figure 3-1* helps you to visualize this. It shows a single directory called *root* as the source of a sample file structure. By descending the branches that extend from root, several other major system directories can be reached. By branching down from these, you can, in turn, reach all the directories and files in the file system. In this hierarchy, files and directories that are subordinate to a directory have what is called a parent/child relationship. This type of relationship is possible for many generations of files and directories, giving you the capability to organize your files in a variety of ways.

YOUR PLACE IN THE FILE SYSTEM STRUCTURE

When you are interacting with the UNIX system, you will be doing so from a location in its file system structure. The UNIX system automatically places you at a specific point in its file system every time you log in. From that point, you can move through the hierarchy to work in any of the directories and files you own and to access those belonging to others that you have permission to use.

The following sections describe your place in relation to the file system structure and how this relationship changes as you move through the file system.

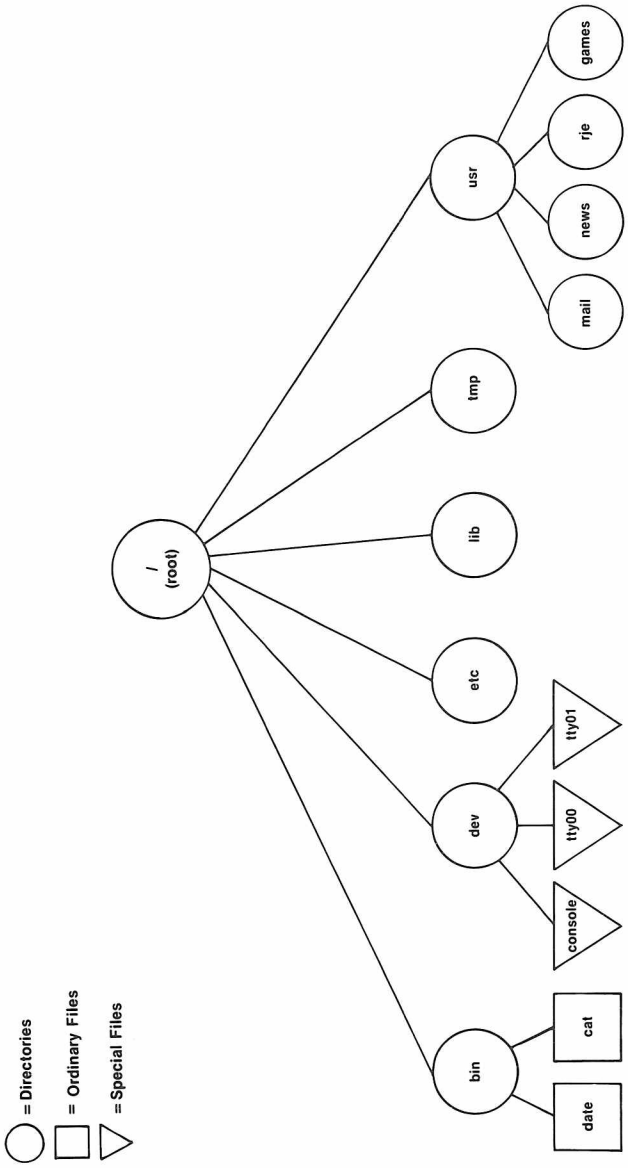


Figure 3-1. Sample file system

Your Home Directory

When you successfully complete the login procedure, the UNIX system positions you at a specific point in its file system structure called your login or home directory. The login name that was assigned to you when your UNIX system account was set up is usually the name of this home directory. In fact, every user with an authorized login name has a unique home directory in the file system.

The UNIX system is able to keep track of all these home directories by maintaining one or more system directories that organize them. For example, let's say that the name of one of these system directories is *user1*, and that it contains the home directories of the login names *starship*, *mary2*, and *jmrs*. *Figure 3-2* shows you how a system directory like *user1* ranks in relation to the other important UNIX system directories you read about in *Chapter 1*.

Within your home directory, you can create files and additional directories (sometimes called subdirectories) to organize them, you can move and delete these files and directories, and you can control who can access your files and directories. You have full responsibility for everything you create in your home directory because you own it. Your home directory is a vantage point from which to view all the files and directories it holds. It is also a point from which to view the file system all the way up to root.

Your Working Directory

As long as you continue to work in your home directory, it is considered your current or working directory. If you move to another directory, that directory becomes your new working directory.

There is a UNIX system command called **pwd**, which stands for **p**rint **w**orking **d**irectory, that you can use to verify the name of the directory in which you are currently working. For example, if your

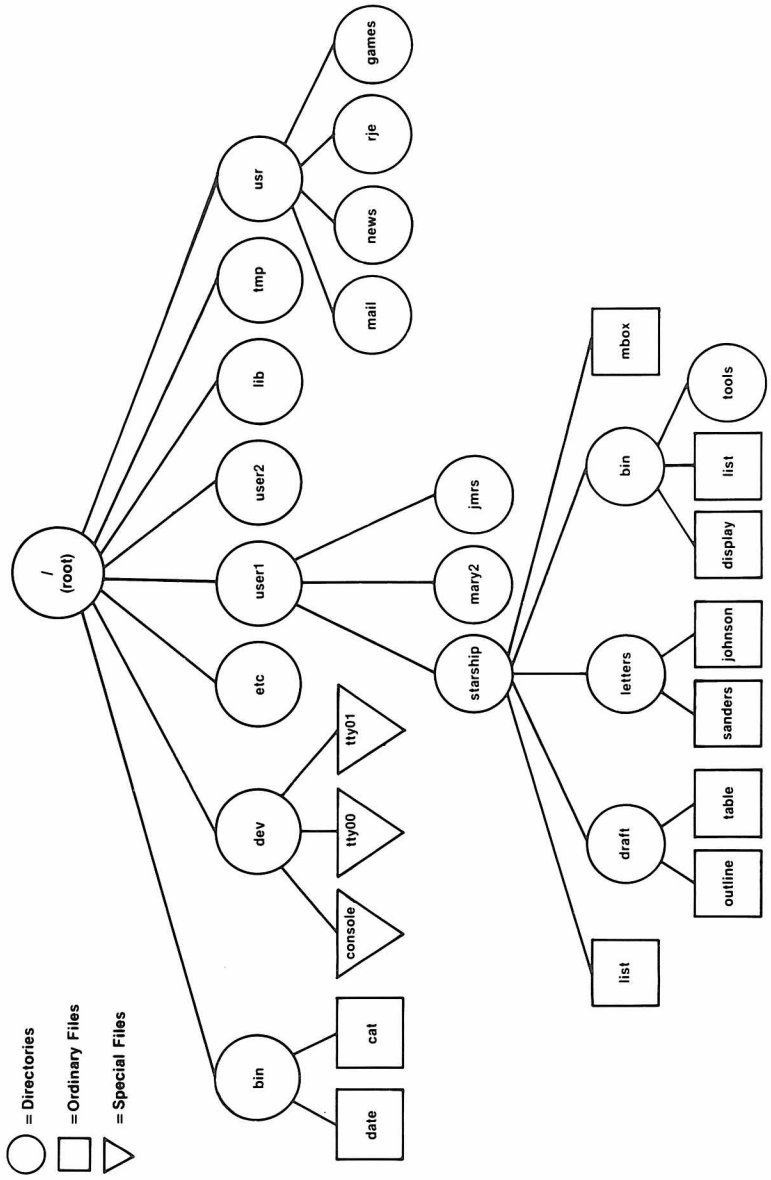


Figure 3-2. A directory that organizes home directories is equivalent to directories like bin and tmp in the file system

login name is *starship* and you issue the **pwd** command in response to the first **$** prompt after logging in, the UNIX system should respond as follows:

```
$ pwd<CR>
/user1/starship
$
```

The system reply indicates that your working directory is */user1/starship*. Technically, */user1/starship* is the full or complete name of the working directory. The name of a directory like */user1/starship* or a file is also referred to as a path name.

Printing the complete or full path name of your working directory in response to a **pwd** command is a courtesy that the UNIX system extends to you. The full path name indicates your exact position in terms of the file system structure.

We will analyze and trace this path name in the next few pages so you can start to move around in the file system. For now, it is sufficient to say that what */user1/starship* tells you is that the root directory **/** (indicated by the leading slash in the line) contains the directory *user1*, which in turn contains the current working directory, which is *starship*. All other slashes in the path name are simply used to separate names of directories and files.

Remember, you are never more than issuing a **pwd** command away from determining where you are in the file system. Issuing the **pwd** command will be especially helpful if you try to read or copy a file and the UNIX system tells you that the file you are trying to access does not exist. You may be surprised to find that you are in a different directory than you thought.

To provide you with a quick summary of what you can expect the **pwd** command to do, a recap of how to use it follows.

Command Recap

pwd - print full name of working directory

command	*options*	*arguments*
pwd	none	none

Description: **pwd** prints the full path name of the directory in which you are currently working.

Remarks: If the system responds with messages, such as, *cannot open directory* or *read error in directory,* there may be problems with the file system. Inform the system administrator.

Path Names

Every file and directory in the UNIX system is identified by a unique path name. The path name tracks or indicates the location of the file or directory relative to the structure of the system. In addition to identifying the location of a file or directory in the file system structure, a path name provides directions to that file or directory. Knowing how to follow the directions the path name gives is your key to moving around the directory structure successfully.

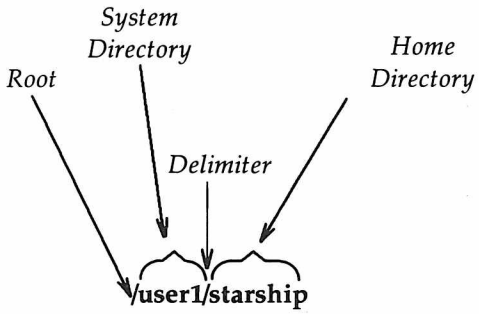
In the file system, there are two types of path names--full and relative. Let's take a closer look at both types.

Full Path Names

A full path name (sometimes called an absolute path name) gives you directions that take you from the root directory down through a unique sequence of directories that leads to a particular directory or file. You can use a full path name to reach any file or directory in the UNIX system in which you are working. A full path name always starts at the root of the file system and its leading character is a / (slash). The final name in a full path name can be either a file name or a directory name. All other names in the path must be directories.

To understand how a full path name is constructed and where it can lead you, let's use the sample file system (*Figure 3-2*) and say that you are in the directory *starship*. If you issue the **pwd** command, the system responds by printing the full path name of your working directory, which is */user1/starship*.

We can analyze the elements of this path name using the following diagram.



where:

/ *(leading)*	=	Root of the file system when it is the first character in the path name,
user1	=	System directory one level below root in the hierarchy to which root points or branches,
/ *(subsequent)*	=	Slash that separates or delimits the directory names, *user1* and *starship*, and
starship	=	Current working directory, which is also the home directory.

Now look at *Figure 3-3*, it traces the full path to */user1/starship* through the sample file system we are using.

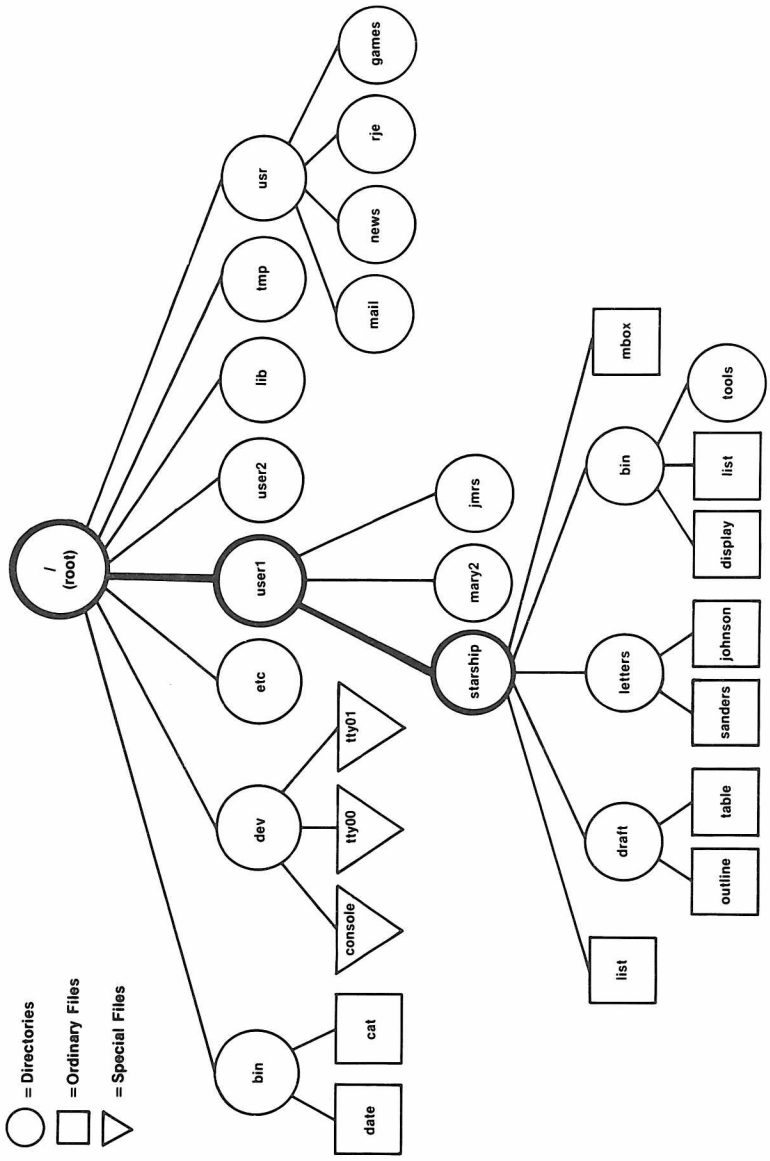


Figure 3-3. Heavy bold lines trace the full path name of the directory /user1/starship

Relative Path Names

A relative path name is the name of a file or directory that varies with relation to the directory in which you are currently working. From your working directory, you can move "down" in the file system structure to access files and directories you own or you can move "up" in the hierarchy through generations of parent directories to the grandparent of all system directories, the root. A relative path name begins with a directory or file name, with a **.** (dot), which is a shorthand notation for the directory in which you are currently located, or a **..** (dot dot), which is a shorthand notation for the directory immediately above your current working directory in the file system hierarchy. The **..** (dot dot) is called the parent directory of the one in which you are currently located, which is the current directory or **.** (dot).

For example, if you are in the home directory *starship* in the sample system and *starship* contains directories named *draft, letters,* and *bin* and a file named *mbox,* the relative path name to any of these is simply its name, be it *draft, letters, bin,* or *mbox. Figure 3-4* traces the relative path name from *starship* to *draft.*

Now, let's say the *draft* directory belonging to *starship* contains the files *outline* and *table.* Then, the relative path name from *starship* to the file *outline* is written as *draft/outline.*

Figure 3-5 traces this relative path. Notice that the slash in this path name separates the directory named *draft* from the file named *outline.* Here, the slash is a delimiter that indicates that *outline* is subordinate to *draft;* that is, *outline* is a child of its parent, *draft.*

Thus far, the discussion of relative path names covered how to specify names and directories of files that belong to, or are children of, your current directory--in other words, to descend the system hierarchy level by level until you reach your destination. You can also, however, ascend the levels in the system structure or ascend and subsequently descend into other files and directories.

To ascend to the parent of your working directory, you can use the **..** notation. This means that if you are in the directory named *draft* in the sample file system, **..** is the path name to *starship,* and **../..** is the path name to *starship*'s parent directory *user1.* From *draft,* you could

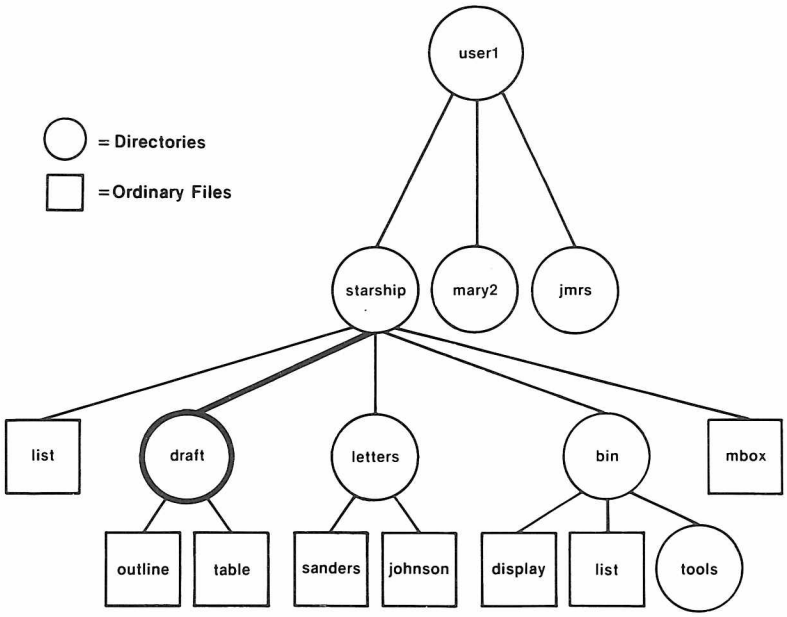


Figure 3-4. Relative path name for the draft directory is traced with heavy bold lines

also trace a path to the file *sanders* in the sample system by using the path name *../letters/sanders* (.. brings you up to *starship*, then down to *letters*, and finally *sanders*).

Keep in mind that you can always use a full path name in place of a relative one.

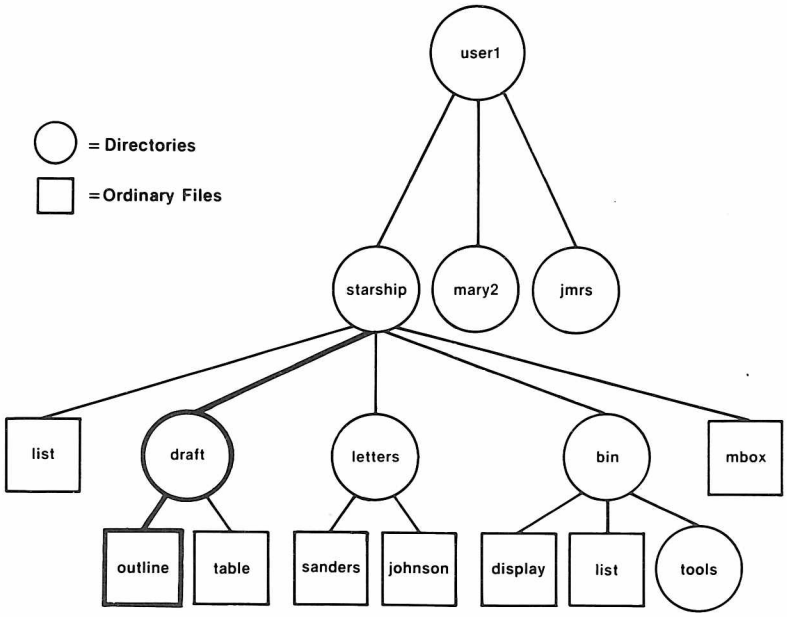


Figure 3-5. The relative path draft/outline is traced in bold lines

In summary, some examples of full and relative path names would be:

Path Name	Meaning
/	Full path name of the root directory for the file system.
/*bin*	Full path name of the *bin* directory that contains most executable programs and utilities.

(continued on next page)

Path Name	Meaning
/user1/starship/bin/tools	Full path name of the directory called *tools* belonging to the directory *bin* that belongs to the directory *starship* belonging to *user1* that belongs to root.
bin/tools	Relative path name to the file or directory *tools* in the directory *bin*. If the current directory is /, then the UNIX system searches for */bin/tools*. But, if the current directory is *starship*, then the system searches the full path */user1/starship/bin/tools*.
tools	Relative path name of a file or directory *tools* in the working directory.

Knowing how to follow path names, such as in these examples, and move about in the file system is a skill tantamount to being able to read and follow a map when you are traveling in a new or unfamiliar place.

It might take some practice to move around in the file system with confidence. But this is to be expected when learning a new concept and the techniques to use it.

To give you a chance to try your hand at moving about in the system's structure, the remainder of the chapter introduces you to the UNIX system commands that make it possible for you to peruse the file system. If you lose track of where you are in the system's structure, use the **pwd** command to identify your location.

ORGANIZING A DIRECTORY STRUCTURE

This section introduces you to four UNIX system commands that make it possible for you to organize and use a directory structure. These commands and what you can expect them to do are as follows:

mkdir -- Allows you to create or make new directories and subdirectories within your current directory,

ls -- Allows you to list the names of all the subdirectories and files in a directory,

cd -- Provides you with the ability to change your location from one directory to another in the file system, and

rmdir -- Lets you remove a directory when you no longer have a need for it.

All of the commands can be used with path names--full or relative--when organizing a directory structure and when moving to the directories and subdirectories you organize, as well as when navigating to directories in the file system that belong to others that you have permission to access. Two of the commands--**ls** and **cd**--can also be used without a path name.

Each of the commands is described more fully in the four sections that follow. In addition, a summary called a command recap is given for each command. The command recaps allow you to review quickly the command line syntax and the capabilities of each command.

Creating Directories *(mkdir)*

It is recommended that you create subdirectories in your home directory according to some logical and meaningful scheme to help you retrieve information you will keep in files. A convenient way to organize your files is to put all files pertaining to one subject together in a directory.

To create a directory, the UNIX system provides you with the **mkdir** command, which stands for **make dir**ectory. In the sample file

system, the *draft* subdirectory in the home directory *starship*, for example, may have been built by inputting the following while located in *starship*:

```
$ mkdir draft<CR>
$
```

The **$** response to the **mkdir** command indicates that a directory named *draft* was successfully created.

Similarly, the other subdirectories named *letters* and *bin* were created with the same command, as indicated in the following screen:

```
$ mkdir letters<CR>
$ mkdir bin<CR>
$
```

All the subdirectories (*draft, letters, bin*) could have been created in one command with the same results, as the following screen shows:

```
$ mkdir draft letters bin<CR>
$
```

You can also move to a subdirectory you created and build additional directories if necessary and reasonable. When you build directories, or create files for that matter, you can name them anything you wish as long as you keep in mind the guidelines presented in the following list.

- The name of a directory (or file) can be from one to fourteen characters in length.
- All characters other than / are legal.
- Some characters are best avoided, such as a blank or space, a tab, or a backspace, and the following:

  @ # $ ^ & * ( ) ` [ ] \ | ; ' " < >

  If you use a blank or tab in a directory or file name, you must enclose the name in quotation marks on the command line.
- Avoid using the +, – or . as the first character in names.
- Uppercase and lowercase characters are distinct to the UNIX system. For example, the directory or file named *draft* would not be the same as the directory or file named *DRAFT*.

Examples of legal directory or file names would be:

memo	MEMO	section2	ref:list
file.c	chap3+4	item1-10	outline

See the command recap that follows for a quick reference to **mkdir**'s capabilities.

Command Recap

mkdir - make a new directory

command	*options*	*arguments*
mkdir	none	**directoryname(s)**

Description: **mkdir** creates a new directory (subdirectory).

Remarks: The system returns the **$** prompt if the directory is successfully created.

Listing the Contents of a Directory *(ls)*

All directories in the file system have information about the files and directories they contain, such as name, size, and the date last modified. You can obtain this information about what your working directory and other system directories contain by using the **ls** command.

The **ls** command, which stands for **list**, lists the names of the files and subdirectories of the directory you specify by path name. If you do not specify a path name, **ls** lists the names of files and directories in your working directory. To demonstrate how the **ls** command works, let's use the sample file system (*Figure 3-2*) once again.

You are logged into the UNIX system and the shell responds to your **pwd** command with the line */user1/starship*. To display the names of files and directories in the working directory, you would type **ls<CR>**. After this sequence, your terminal should read:

```
$ pwd<CR>
$ /user1/starship
$ ls<CR>
bin
draft
letters
list
mbox
$
```

As you can see, the system responds by listing the names of files and directories in the working directory *starship* in alphabetical order. If the first character of any of the file or directory names was a number, or a capital letter, it would have been printed first.

Now, if you want to print the names of files and subdirectories in a directory other than your working directory without moving from your working directory, you should use the command format:

ls directoryname<CR>

where the directory name is the full or relative path name of the desired directory. This means that you can print the contents of *draft* while you are working in *starship* by inputting **ls draft<CR>**.

```
$ ls draft<CR>
outline
table
$
```

In the example, *draft* is a relative path name from *starship* to *draft*. By the same token, you could print the contents of the *user1* directory, which is the parent of the *starship* by typing:

```
$ ls ..<CR>
jmrs
mary2
starship
$
```

where .. is the relative path name from *starship* to *user1*. You could also list the contents of *user1* by typing **ls /user1<CR>** (since */user1* is the full path name from root to *user1*) and get the identical listing.

Similarly, you can list the contents of any system directory that you have permission to access using the **ls** command and a full or relative path name.

The **ls** command is particularly useful if you have a long list of files and you are trying to determine whether one of them exists in your working directory. For example, if you are in the directory *draft* and you wish to determine if the files named *outline* and *notes* are there, you can use the **ls** command as follows:

```
$ ls outline notes<CR>
outline
notes not found
$
```

The output on the terminal monitor shows that the system acknowledges the existence of *outline* by printing its name, but says that the file *notes* is not found.

By the way, the **ls** command will not print the contents of a file. If you wish to see what a file contains, you can use the **cat**, **pg**, or **pr** command, which are described in the section of this chapter entitled *Accessing and Manipulating Files.*

Frequently Used ls Options

The **ls** command also accepts options that cause specific attributes of a file or subdirectory to be listed. There are more than a dozen available options for the **ls** commands. Of these, the **−a** and **−l** will probably be most valuable in your basic use of the UNIX system. Refer to the *UNIX System User Reference Manual* for information and details on the other options.

Listing All Names in a File. Some important file names in your home directory begin with a **.** (dot), such as .profile, **.** (the current directory), and **..** (the parent directory). The **ls** command will not

print these names unless you use the **−a** option in the command line. Thus, to list all files in your working directory *starship*, including those that start with a **.** (dot), type **ls −a<CR>**. The terminal should look something like this:

```
$ ls −a<CR>
.
..
.profile
bin
draft
letters
list
mbox
$
```

Listing Contents in Long Format. Probably the most informative **ls** option is **−l**. If you type **ls −l<CR>** while in the *starship* directory, you would get the following:

```
$ ls −l<CR>
total 30
drwxr-xr-x  3  starship project    96  Oct 27   08:16 bin
drwxr-xr-x  2  starship project    64  Nov  1   14:19 draft
drwxr-xr-x  2  starship project    80  Nov  8   08:41 letters
-rwx------  2  starship project 12301  Nov  2   10:15 list
-rw-------  1  starship project    40  Oct 27   10:00 mbox
$
```

After the command line, the first line of output, *total 30*, shows the amount of memory used, which is measured in chunks called blocks. Next is one line for each directory and file. The first character in each of these lines tells you what kind of file is listed, where:

d = Directory,

– = Ordinary disk file,

b = Block special file, and

c = Character special file.

The next several characters, which are either letters or hyphens, describe who has permission to read and use the file or directory. (Permissions are discussed with the **chmod** command in the section entitled *Accessing and Manipulating Files* in this chapter.) The following number is the link count, which in the case of a file, equals the number of directories it is in, or in the case of a directory, also includes the number of directories immediately under it in the file system structure. Next is the login name of the owner of the file, which is *starship*, and then the group name of the file or directory, which is *project*. The following number indicates the length of the file or directory entry measured in units of information (or memory) called bytes. Then there is the month, day, and time that the file was last modified. Finally, the file or directory name is given.

Figure 3-6 sums up what you get when you list the contents of a directory in long format.

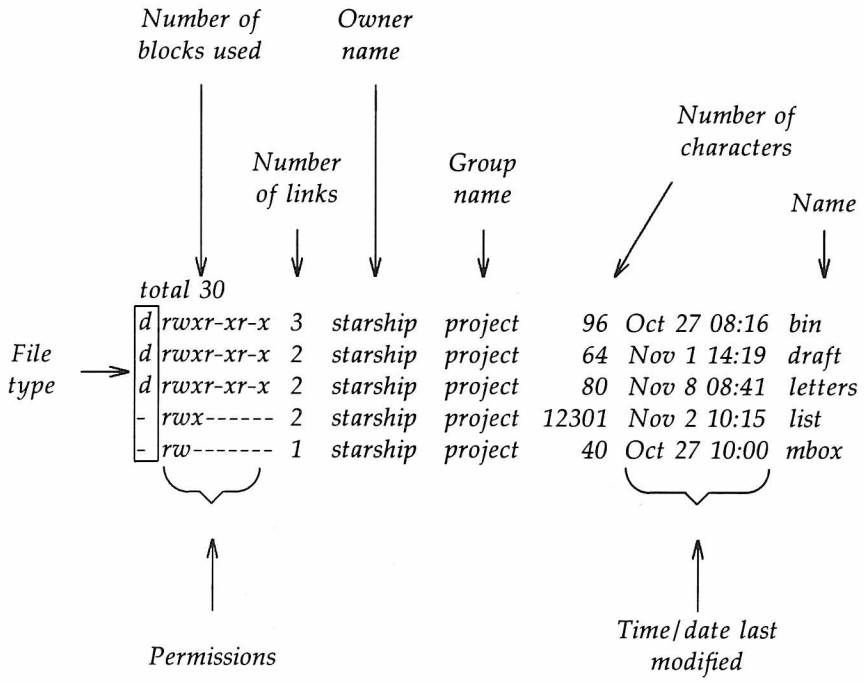


Figure 3-6. Description of output produced by the ls −l command

Command Summary. Following is a recap of capabilities provided by the **ls** command and two available options. See the *UNIX System User Reference Manual* for information on other available options.

Command Recap

ls - list contents of a directory

command	*options*	*arguments*
ls	**−a**, **−l**, and others*	**directoryname(s)**

Description: **ls** lists the names of the files and subdirectories in the specified directories. If no directory name is given as an argument, the contents of your working directory are listed.

Options:

−a Lists all entries, including those beginning with **.** (dot).

−l Lists contents of a directory in long format furnishing mode, permissions, size in bytes, and time of last modification.

Remarks: If you want to read the contents of a file, use the **cat** command.

* See the *UNIX System User Reference Manual* for all available options and an explanation of their capabilities.

Changing Your Working Directory *(cd)*

When you first log into the UNIX system, you are placed in your home directory, which becomes your current or working directory. You may, however, wish to work in a different directory for any number of reasons. For example, you might want to create a file in a specific directory, you may need to make corrections to a file in another directory, or you may wish to obtain information by reading a file in a different directory.

Whatever the reason, the UNIX system provides you with the **cd** command that allows you to move around in its directory structure. When you use the **cd** command to move to a new directory, that directory becomes your working directory.

To use the **cd** command, enter the command:

cd newdirectory-pathname<CR>

where the path name, whether full or relative, to the new directory is optional. Any valid path name of a directory can be used as an argument to the **cd** command. If you use the **cd** command without specifying a path name, it will move you to your login directory regardless of where you are in the file system.

When you specify a valid directory path name on the command line, the UNIX system moves you to that directory. For example, to move from the *starship* directory to the child directory *draft* in the sample file system, type **cd draft<CR>**. In this example, *draft* is the relative path name to the desired directory. When you get the **$** prompt, verify your new location by typing **pwd<CR>**. Your terminal monitor should look something like the following after going through this sequence:

```
$ cd draft<CR>
$ pwd<CR>
/user1/starship/draft
$
```

Now that you are in the *draft* directory you can access the files and directories in it, in this case, the files *outline* and *table*. You can also create subdirectories in *draft* with **mkdir** and additional files with the **ed** and **vi** commands. (See *Chapter 4* for general information on the **ed** and **vi** commands and *Chapter 5* and *Chapter 6* for tutorials on using the **ed** and **vi** commands, respectively.)

You may also use full path names with the **cd** command. For example, to move to the *letters* directory from the *draft* directory, you could use the command

cd /user1/starship/letters<CR>

where */user1/starship/letters* is the full path name to *letters*.

Or, since *letters* and *draft* are both children of *starship*, you could use the **cd** command with the relative path name *../letters*. The **..** notation moves you to the directory *starship*, and the remainder of the path name moves you to *letters*.

If you wish to return to your home directory after perusing the file system, simply type **cd<CR>**. The **cd** command with no arguments returns you to your login directory.

Command Recap

cd - change your working directory

command	*options*	*arguments*
cd	none	**directoryname**

Description: **cd** changes your position in the file system from the current directory to the directory specified. If no directory name is given as an argument, the **cd** command places you in your home directory.

Remarks: When the shell places you in the directory specified, the **$** prompt is returned to you. You will also receive a **$** prompt when you issue the **cd** command with no argument. To access a directory that is not in your working directory, you must substitute the full or relative path name in place of a simple directory name.

Removing Directories *(rmdir)*

If you decide you no longer need a directory, you can remove it with the **rmdir** command. The **rmdir** command, which stands for **rem**ove a **dir**ectory, removes a directory if that directory does not contain subdirectories and files, or, in other words, if the directory is empty.

If the directory you are attempting to remove is not empty, **rmdir** will not remove it unless you remove the contents of the directory first. In addition, you are not allowed to remove directories belonging to other system users unless you have permission to do so.

The standard format for the **rmdir** command is:

rmdir directoryname(s)<CR>

where one or more directory names can be specified.

If you were to attempt to remove the directory *bin* in the sample file system, the UNIX system would respond in the following manner:

```
$ rmdir bin<CR>
rmdir: bin not empty
$
```

To remove the directory *bin* with the **rmdir** command, you would first have to remove the files *display* and *list* and the subdirectory *tools*. If you wish to remove files, see the section entitled *Accessing and Manipulating Files* in this chapter. To remove any subdirectories like *tools*, use the **rmdir** command. The system will return the **$** prompt in response to the **rmdir** command when the directory specified in the command line is empty.

The command recap that follows summarizes how **rmdir** works.

Command Recap

rmdir - remove a directory

command	*options*	*arguments*
rmdir	none	**directoryname(s)**

Description: **rmdir** removes named directories if they do not contain files and/or subdirectories.

Remarks: If the directory is empty, the system returns the **$** prompt when the directory is removed. If the directory contains files or subdirectories, the message, *rmdir: directory name not empty* is returned to you.

ACCESSING AND MANIPULATING FILES

This section introduces you to several UNIX system commands that access and manipulate files in the file system structure. Information in this section is organized into two parts--basic and advanced. The part devoted to basic commands is fundamental to your using the file system; the advanced commands offer you more sophisticated information processing techniques when working with files. You may skip reading the advanced section if you do not need to use the commands it covers.

Basic Commands

This section discusses UNIX system commands that are important to your being able to access and use the files in your directory structure. Specifically, these commands and their capabilities are:

cat -- Outputs the contents of a file you name,

pg -- Prints on a video display terminal the contents of a file you name in chunks or pages,

pr -- Prints on your terminal a partially formatted version of the file you name,

lp -- Allows you to request a paper copy of a file from a device called the line printer,

cp -- Makes a duplicate copy of an existing file,

mv -- Moves and renames a file,

rm -- Permanently removes a file when you no longer need it,

wc -- Counts the lines, words, and characters in a file, and

chmod --Changes permission modes for a file (and a directory).

Each command is covered in one of following sections. A command recap follows the discussion of each command allowing you to review quickly the command line syntax and command capabilities.

Displaying a File's Contents (cat, pg, pr)

The UNIX system provides three commands that allow you to display and print the contents of a file or files--**cat, pg,** and **pr**. The **cat** command, which stands for con**cat**enate, outputs the contents of files you specify by name on the command line, and displays the result on your terminal unless you tell **cat** to direct the output to another file or a new command. The **pg** command is particularly useful when you wish to read the contents of a lengthy file or a number of files because the command displays the text of a file in chunks or pages, a screenful at a time at your direction on a video display terminal. The **pr** command partially formats and outputs the files you specify on your terminal unless you direct the output to a paper printing device (see the **lp** command in this chapter).

The following three sections describe how to use these commands.

Concatenate and Print Contents of a File *(cat).* The **cat** command displays the contents of a file or files. For example, if you are located in directory *letters* in the sample file system and you wish to display the contents of the file *johnson,* you would type **cat johnson<CR>** and the following output would appear on the terminal.

```
$ cat johnson<CR>
This file contains a letter
to Mr. Johnson on the topic of
office automation.
$
```

As you can see, the contents of the file are displayed after the command line and are followed by the **$** prompt.

To display the contents of two (or more) files, like *johnson* and *sanders,* simply type **$ cat johnson sanders<CR>** and the **cat** command reads *johnson* and *sanders* and displays their contents in that order on your terminal.

```
$ cat johnson sanders<CR>
This file contains a letter
to Mr. Johnson on the topic of
office automation.
This file contains a letter
to Mrs. Sanders inviting her to
speak at our departmental
meeting.
$
```

To direct the output of the **cat** command to another file or to a new command, see the section in *Chapter 7* that discusses redirecting input and output.

The command recap that follows summarizes what you can expect the **cat** command to do.

Command Recap

cat - concatenate and print a file's contents

command	*options*	*arguments*
cat	available*	**filename(s)**

Description: **cat** reads the name of each file given on the command line and displays the contents of the files.

Remarks: If the file(s) exist, the contents are displayed on the terminal monitor; if not, the message *cat: cannot open filename* is returned to you.

If you wish to display the contents of a directory, use the **ls** command.

* See the *UNIX System User Reference Manual* for all available options and an explanation of their capabilities.

Paging Through the Contents of a File *(pg)*. The **pg** command, short for **page**, allows you to examine the contents of a file or files screenful by screenful on a video display terminal. The **pg** command displays the text of a file in chunks or pages followed by a colon (:). After displaying the colon, the system pauses and waits for your instructions to proceed. For example, your instructions can request **pg** to continue displaying the file's contents a page at a time or you can ask **pg** to search through the file(s) to locate a specific character pattern. *Table 3-1* summarizes some of the instructions you can give **pg** after the colon is displayed.

TABLE 3-1
Summary of Selected Commands for pg*

Command†	Meaning
h	Help; display list of available **pg** commands
q *or* Q	Quit **pg** perusal mode
<CR>	Display next page of text
l	Display next line of text
d or ^d	Display additional half page of text
. or ^l	Redisplay current page of text
f	Skip next page of text, and display following one
n	Begin displaying next file you specified on command line
p	Display previous file specified on command line
$	Display last page of text in file currently displayed
/pattern/	Search forward in file for specified character pattern
^pattern^	Search backward in file for specified character pattern

* See the *UNIX System User Reference Manual* for a detailed explanation of all available **pg** commands.

† Most commands can be typed with a number preceding them: +1 (display next page), −1 (display previous page), or 1 (display first page of text).

The **pg** command is especially useful when you wish to peruse a long file or a series of files because the system pauses after displaying each page allowing you as much time as you need to examine it. The size of the page displayed depends on the terminal you are using. For example, on a video display terminal with a window capable of showing 24 lines, 23 lines of text and a line containing the colon will be displayed as a page. However, if the file is less than 23 lines long, the page size will be the number of lines in the file plus the line containing the colon.

To peruse the contents of a file with **pg**, use the following command line format:

pg filename(s)<CR>

For example, to display the contents of the file **outline** in the sample file system, type **pg outline<CR>** and the first page of the file will appear on the screen. Since the file has more lines in it than can be displayed in one page, the colon indicates there is more to be looked at when you are ready. Pressing the **<CR>** key will print the next page of the file.

The following screen summarizes what has been done thus far.

```
$ pg outline<CR>
After you analyze the subject for your
report, you must consider organizing and
arranging the material you wish to use in
writing it.
        .
        .
        .
An outline is an effective method of
organizing the material. The outline
is a type of blueprint or skeleton,
a framework for you the builder-writer
of the report; in a sense it is a recipe
:<CR>
```

After pressing the **<CR>** key, the **pg** program will resume outputting the file's contents on the screen as follows:

```
that contains the names of the
ingredients and the order in which
to use them.
      .
      .
      .
Your outline need not be elaborate or
overly detailed; it is simply a guide you
may consult as you write, to be varied,
if need be, when additional important
ideas are suggested in the actual writing.
(EOF):
```

In addition to the remainder of the file's contents, a line with the output *(EOF):* is displayed. The EOF designates that you have reached the end of the file and the colon is your cue for the next instruction.

When you have completed examining the file, you can type **q** or **Q** followed by pressing the **<CR>** key and the **$** prompt will appear on your screen. Or you can choose to use one of the other available commands given in *Table 3-1* depending on your needs.

In addition, there are a number of options that can be specified on the **pg** command line. Refer to the *UNIX System User Reference Manual* if you are interested in learning more about them.

The following command recap summarizes the highlights of **pg**'s capabilities.

Command Recap

pg - display a file's contents in chunks or pages

command	*options*	*arguments*
pg	available*	**filename(s)**

Description: **pg** reads the name of each file given on the command line and displays the contents of the file(s) in chunks or pages, screenful by screenful.

Remarks: After displaying a screenful of text, the **pg** command awaits your instruction to continue to display text, to search for a pattern of characters, or to exit the **pg** perusal mode. In addition, a number of options are available for you to use with **pg** on the command line. For example, you can start to display the contents of file at a specific line or at a line containing a certain sequence or pattern or you can opt to go back and review text that has already been displayed.

* See the *UNIX System User Reference Manual* for all available options and an explanation of their capabilities.

Print Partially Formatted Contents of a File *(pr)*. The **pr** command is typically used to prepare files for printing. You can expect the **pr** command to title, paginate, supply headings, and print a file according to varying page lengths and widths on your terminal monitor unless you specify that it prints on another output device, such as a line printer (read the discussion on the **lp** command in this section), or you direct the printing to a different file (see the section on redirecting input and output in *Chapter 7)*.

If you choose not to specify any of the available options, the **pr** command produces output that is in a single column with 66 lines per page and is preceded by a short heading. The heading consists of five lines--two blank lines; a line containing the date, time, file name, and page number; and two more blank lines. And the formatted file is followed by five blank lines. (Complete sets of text formatting

tools, called **nroff** and **troff**, are available on UNIX systems equipped with the appropriate application software. Check with your system administrator to see if this software is available to you.)

Typically, the **pr** command is used in tandem with the **lp** command to provide a paper copy of text as it was entered into a file. (See the section discussing the **lp** command for details.) However, you can also use the **pr** command to format partially and print the contents of a file on your terminal. For example, to review the contents of the file *johnson* in the sample file system, type in the command **pr johnson<CR>**. The following screen summarizes this activity.

```
$ pr johnson<CR>

Nov 29 09:19 1983  johnson Page1

This file contains a letter
to Mr. Johnson on the topic of
office automation.
.
.
.
$
```

Note that the ellipses after the last line in the file stand for the remaining 58 lines (all blanks in this case) that **pr** formatted into the output. If you are working on a video display terminal, which typically allows you to view about 24 lines at a time, the entire 66 lines of the formatted file will print continuously and rapidly to the end of file. This means that the first 41 lines will "roll" off the top of your screen making it impossible for you to read them unless you have the ability to "roll" or "page" back a screen or two. If you are looking at a particularly long file, this feature might not solve the problem.

In this case, you should use the control-s <^s> combination to stop printing on your terminal temporarily and control-q <^q> to resume the printing.

The command recap that follows summarizes what you can expect the **pr** command to do.

Command Recap

pr - print partially formatted contents of a file

command	*options*	*arguments*
pr	available*	**filename(s)**

Description: **pr** produces a partially formatted copy of a file(s) on your terminal monitor unless otherwise specified. The program prints the text of the file(s) on 66-line pages and places five blank lines at the bottom of each page and a five-line heading at the top of each page. The heading consists of two blank lines; a line containing the date, time, file name and page number; and two additional blank lines.

Remarks: If the specified file(s) exists, the contents are partially formatted and displayed on the screen; if not, the message *pr: can't open filename* is returned to you.

The **pr** command is most commonly used with the **lp** command when a paper copy of a file is needed. However, when using the **pr** command to review a file on a video display terminal, use <^s> and <^q> to temporarily stop and start printing the file.

* See the *UNIX System User Reference Manual* for all available options and an explanation of their capabilities.

Requesting a Paper Copy of a File (lp)

At some point in time, you may want a paper copy of a file. Some terminals have built-in printers that allow you to get paper copies of files. In this case you simply need to turn the printer on and then use **cat** or **pr** to print the file. If, however, you wish to obtain a higher quality paper copy, you should consider using the **lp** command. The **lp** command, which stands for **l**ine **p**rinter, allows you to request a line-printing device to furnish you with a paper copy of a file or files (*Figure 3-7*).

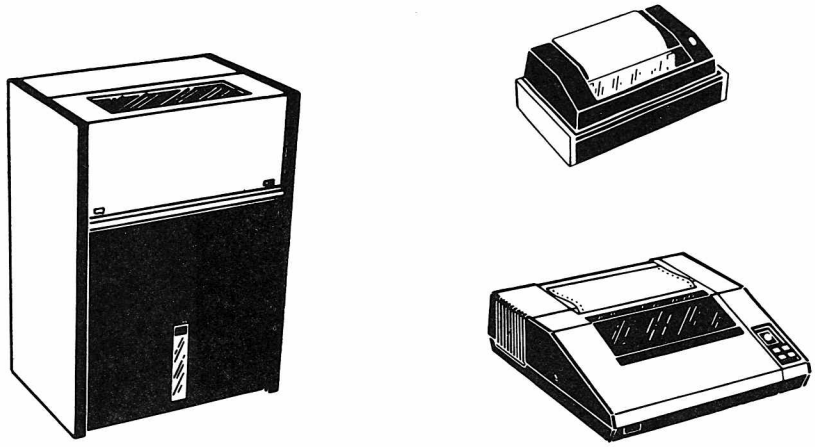


Figure 3-7. Examples of TELETYPE Model 40 line printers; left, printer with tractor feed belt; upper right, printer with tractor feed; bottom right, printer with high-speed tractor feed

The line printer or types of line printers that you have access to depends on what your UNIX system facility has to offer. You should ask your system administrator for the names of the line printers

available to you. Or you can type **lpstat −v<CR>** to obtain a complete listing of every accessible line-printing device.

The basic format for the command is:

lp file<CR>

For example, to print the file *letters* on a line printer, you would type **lp letters<CR>** on the command line. In turn, the system would provide you with the name of the device or type of device on which the file will be printed and an identification (id) number indicating your request. The following screen summarizes this activity.

```
$ lp letters<CR>
Request id is laser-6885   1 file
$
```

The system response indicates that your job is to be printed on a laser line-printing device (the system default), has a request id number of 6885, and is to include the printing of one file.

Using the **−d***dest* (*destination*) option on the command line would cause your file to be printed on another available device that you name in place of *dest*. Using the **−m** option would cause mail to be sent to you indicating when the job is completed.

If you would like to cancel the request to **lp** to print the file *letters*, type **cancel laser-6885<CR>**, where laser-6885 is the request id. The **lpstat** command gives the status and request id of the line printer jobs.

A command recap follows that summarizes what you can expect of the **lp** command.

Command Recap

lp - request paper copy of file from a line printer

command	*options*	*arguments*
lp	**−d, −m,** and others*	**file(s)**

Description: **lp** requests that specified files be printed by a line printer, thus providing paper copies of the contents.

Options:

−d*dest* Allows you to choose *dest* as the printer or type of printer that is to produce the paper copy. If you do not use this option, the **lp** program specifies the printer for you.

−m Sends a message to you via **mail** after the printing is complete.

Remarks: You can cancel a request to the line printer by typing **cancel** and the request **id** furnished to you by the system when the request was acknowledged.

Check with the system administrator for information on additional and/or different commands for printers that may be available at your location.

* See the *UNIX System User Reference Manual* for all available options and an explanation of their capabilities.

Making a Duplicate Copy of a File (cp)

When using the UNIX system, you may wish to make a copy of a file. For example, you might want to revise a file while leaving the original version intact. The UNIX system provides you with the **cp** command, short for **copy**, which copies the complete contents of one file into another. The **cp** command also allows you to copy one or

more files from one directory into a different directory while leaving the original file or files in place.

To copy the file named *outline* to a file named *new.outline* in the sample directory, simply type **cp outline new.outline<CR>**. The system returns the **$** prompt when the copy is made. To verify the existence of the new file, you can type **ls<CR>**, which lists the names of all files and directories in the current directory, in this case *draft*. The following screen summarizes the activity.

```
$ cp outline new.outline<CR>
$ ls<CR>
new.outline
outline
table
$
```

You know from looking at the sample file system that the file *new.outline* did not exist before the **cp** command to copy *outline* to *new.outline* was given. However, if it had, it would have been replaced by a copy of the file *outline* and the previous version of *new.outline* would have been deleted.

If you had tried to copy the file *outline* to another file named *outline* in the same directory, the system would have told you that the file names were identical and returned the **$** prompt to you. If you listed the contents of the directory to determine exactly how many copies of *outline* exist, the terminal monitor would look something like the following:

```
$ cp outline outline<CR>
cp: outline and outline are identical
$ ls<CR>
outline
table
$
```

As you can see, the UNIX system does not allow you to have two files with the same name in a directory.

You could, however, copy the file named *outline* from the directory *draft* to another file named *outline* in the directory named *letters* by using any of the following command lines assuming you are currently in *draft*:

```
cp outline ../letters/outline<CR>
cp outline ../letters<CR>
cp outline /user1/starship/letters/outline<CR>
cp outline /user1/starship/letters<CR>
```

A copy of the file *outline* would reside in both directories *draft* and *letters* after using one of these commands since each of them contains a legal path name to the file *outline*. From this example, you can see that the UNIX system allows you to have files with identical names as long as they are in different directories.

If you would like to copy the file *outline* in the directory *draft* to a file named *outline.vers2* in the directory *letters*, you could use either of the following command lines:

```
cp outline ../letters/outline.vers2<CR>
cp outline /user1/starship/letters/outline.vers2<CR>
```

Keep in mind the conventions for naming directories and files given in the section entitled *Creating Directories* in this chapter.

The following recap summarizes how the **cp** command works.

Command Recap

cp - make a copy of a file

command	*options*	*arguments*
cp	none	**file1 file2** **file(s) directory**

Description: **cp** allows you to make a copy of *file1* and call it *file2* leaving *file1* intact, or to copy one or more files into a different directory.

Remarks: When copying *file1* to *file2* and *file2* already exists, the **cp** command will overwrite the first version of *file2* with a copy of *file1* calling it *file2*. The first version of *file2* is deleted.

You cannot copy directories with the **cp** command.

Moving and Renaming a File (mv)

The **mv** command allows you to rename a file in the same directory or to move a file from one directory to another. If you move a file to a different directory, the file can be renamed or it can retain its original name.

To rename a file in a directory, use the following command:

mv file1 file2<CR>

The **mv** command changes a file's name from *file1* to *file2*. Remember that the names *file1* and *file2* can be any valid names, including path names.

For example, if you are in the directory *draft* in the sample file system and you would like to rename the file *table* as *new.table*, simply type **mv table new.table<CR>**. You should receive the **$** command

prompt if the command executed successfully. To verify that the file *new.table* exists, you can list the contents of the directory by typing **ls<CR>**. In turn, the terminal should read:

```
$ mv table new.table<CR>
$ ls<CR>
new.table
outline
$
```

You can also move a file from one directory to another keeping the file's name the same or changing it to a different one. To do so, use the following command line.

mv file(s) directory<CR>

where the file and directory names can be any valid names, including path names.

To move the file *table* from the current directory named *draft* (whose full path name is */user1/starship/draft*) to a file with the same name in the directory **letters** (whose relative path name from *draft* is *../letters* and whose full path name is */user1/starship/letters*), any one of several command lines can be used, including the following:

```
mv table /user1/starship/letters<CR>
mv table /user1/starship/letters/table<CR>
mv table ../letters<CR>
mv table ../letters/table<CR>
mv /user1/starship/draft/table /user1/starship/letters/table<CR>
```

The file *table* could have been renamed *table2* when moving it to the directory *letters* using any of the following:

```
mv table /user1/starship/letters/table2<CR>
mv table ../letters/table2<CR>
mv /user1/starship/draft/table2 /user1/starship/letters/table2<CR>
```

You can verify that the command worked by listing the contents of the directory with the **ls** command.

Refer to the recap that follows for a summary of how the **mv** command works.

Command Recap

mv - move or rename files

command	*options*	*arguments*
mv	none	**file1 file2** **file(s) directory**

Description: **mv** allows you to change the name of a file or to move a file(s) into another directory.

Remarks: When changing the name of *file1* to *file2* and *file2* already exists, the **mv** command will overwrite the first version of *file2* with *file1* and rename it *file2*. The first version of *file2* is deleted.

Removing a File (rm)

When you no longer need a file, you can get rid of it by using the **rm** command, which is short for **remove**.

To remove one or more files, use the format:

rm file(s)<CR>

After the command executes, the file(s) you specified are removed permanently.

To remove a file named *new.outline* in the current directory type **rm new.outline<CR>** and list the contents of the directory with the **ls** command to verify that the file *new.outline* no longer exists.

To remove more than one file, such as the files *outline* and *table*, type **rm outline table<CR>** and list the contents of the directory by typing **ls<CR>**.

```
$ rm outline table<CR>
$ ls<CR>
$
```

The **$** response indicates that the files named *outline* and *table* were removed permanently.

The following recap summarizes how the **rm** command works.

Command Recap

rm - remove a file

command	*options*	*arguments*
rm	available*	**file(s)**

Description: **rm** allows you to remove one or more files.

Remarks: Files specified as arguments to the **rm** command are removed permanently.

* See the *UNIX System User Reference Manual* for all available options and an explanation of their capabilities.

Counting Lines, Words, and Characters in a File (wc)

The **wc** command, which stands for **w**ord **c**ount, reports the number of lines, words, and characters there are in a file that you specify by name on the command line. If you name more than one file, the **wc**

program counts the number of lines, words, and characters in each specified file and then totals the counts. In addition, you can direct the **wc** program to give you only a line, a word, or a character count by using the **−l**, **−w**, or **−c** options, respectively.

To determine the number of lines, words, and characters in a file, use the following format on the command line:

wc file1<CR>

When you do, the system responds with a line in the format:

l w c file1

where

l = Number of lines in *file1*,

w = Number of words in *file1*, and

c = Number of characters in *file1*.

For example, to count the lines, words, and characters in the file *johnson* in the current directory *letters*, type **wc johnson<CR>**. The terminal monitor would show the following output:

```
$ wc johnson<CR>
    3    14    78  johnson
$
```

The system response displays the line count (*3*), the word count (*14*), and the character count (*78*) for the file *johnson*.

To determine the number of lines, words, and characters in more than one file, use the following format:

wc file1 file2<CR>

In turn, the system responds with the following format:

l	*w*	*c*	*file1*
l	*w*	*c*	*file2*
l	*w*	*c*	*total*

where line, word, and character counts are displayed for *file1* and *file2* on separate lines and the combined counts appear on the last line called *total.*

If you request that the **wc** program count lines, words, and characters in the files *johnson* and *sanders* in the current directory, the system would respond as follows:

```
$ wc johnson sanders<CR>
     3    14    78  johnson
     4    16    95  sanders
     7    30   173  total
$
```

In this case, the first line of the system response shows the line, word, and character counts for the file *johnson.* The second line of output gives line, word, and character counts for *sanders.* The last line of output shows combined line, word, and character counts for both files in the line labeled *total.*

If you prefer to get only a line, a word, or a character count, select the appropriate format from the following lines:

wc	**−l**	**file1<CR>**	(line count)
wc	**−w**	**file1<CR>**	(word count)
wc	**−c**	**file1<CR>**	(character count)

For instance, by typing **wc −l sanders<CR>** on the command line you would obtain the following output:

```
$ wc -l sanders<CR>
      4  sanders
$
```

The system tells you that the number of lines in the file *sanders* is 4 in answer to specifying **−l**. If the **−w** or **−c** option was specified for that file, the UNIX system would have responded with the number of words or number of characters, respectively, in the file.

The command recap that follows summarizes how the **wc** command works.

Command Recap

wc - count lines, words, and characters in a file

command	*options*	*arguments*
wc	**−l, −w, −c**	**file(s)**

Description: **wc** counts lines, words, and characters in the file(s) named keeping a total count of all tallies when more than one file is specified.

Options

−l Counts the number of lines in the specified file(s).

−w Counts the number of words in the specified file(s).

−c Counts the number of characters in a specified file(s).

Remarks: When a file name is specified in the command line, it is printed with the count(s) requested.

Protecting Your Files *(chmod)*

The **chmod** command, short for **ch**ange **mod**e, allows you to decide who can read, alter, and use your files and who cannot. Because the UNIX operating system is a multiuser system, you are not working alone in the file system: you and other system users can follow path names and run system commands to move to various directories and to read and use files belonging to one another if you have permission to do so.

If you own a file, then you are able to determine who has the right to read that file, to make changes to or write the file, and to run or execute the file if it is a program. These permissions are defined as:

r = Allows system users to read a file or to copy its contents,

w = Allows system users to write changes into a file or copy of a file, and

x = Permits system users to run an executable file.

Specifically, you can determine who in the population of UNIX system users is entitled to these various permissions and who is not according to the following classifications:

u = You, the user and login owner of your files and directories,

g = Members of the group to which you belong (the group could consist of team members working on a project, members of a department, or a group arbitrarily designated by the person who set up your UNIX system account), and

o = All other system users.

When you create a file or a directory, the system automatically grants or denies permission specifically to you, members of your group, and other system users. You can alter this automatic action to some extent by modifying your environment, which is discussed in *Chapter 7*. Regardless of how the permissions are granted when a file is created, as the owner of the file or directory it is up to you to allow current

permissions to remain in effect or to change them to suit your purposes and the situation. For example, you may wish to keep certain files private and for your use only. Or you may wish to grant permission to read and to write changes into a file to members of your group and all other system users as well. Or you may share a program with members of your group by granting them permission to execute it.

How to Determine Existing Permissions. You can determine what permissions are currently in effect on a file or a directory by using the command that produces a long listing of a directory's content, which is **ls −l**. For example, typing **ls −l<CR>** while in the directory named *starship/bin* in the sample file system would produce the following output:

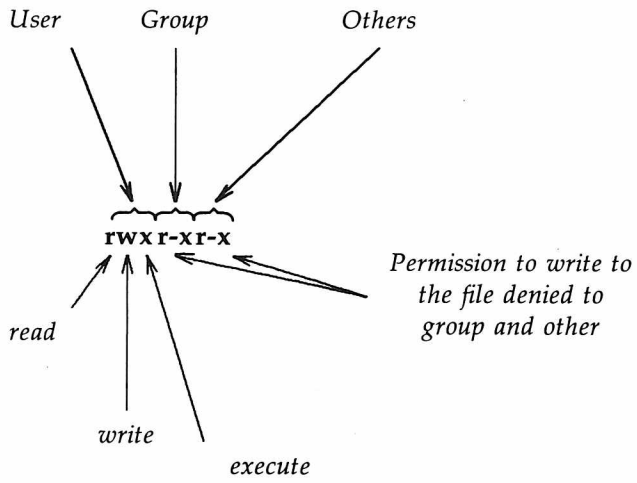
```
$ ls −l<CR>
total 35
-rwxr-xr-x   1  starship project   9346  Nov 1  08:06  display
-rwx--x--x   1  starship project   6428  Dec 2  10:24  list
drwx--x--x   2  starship project     32  Nov 8  15:32  tools
$
```

Permissions for the files *display* and *list* and the directory *tools* are shown on the left of the terminal monitor under the line *total 35*, and look like:

rwxr-xr-x	(file *display*)
rwx--x--x	(file *list*)
rwx--x--x	(directory *tools*)

These nine characters represent three groups of three characters. The first set of three characters refers to your (or the user's/owner's) permissions, the second set to members of the group, the last set to all other system users. Within each set of characters, the *r*, *w*, and *x* indicate the permission currently enabled for the groups. If a dash appears instead of an *r*, *w*, or *x*, permission to read, write, or execute is denied.

The following diagram summarizes this breakdown for the file named *display*.



As you can see, the owner has *r*, *w*, and *x* permissions and members of the group and other system users have *r* and *x* permissions.

How to Change Existing Permissions. After you have determined what permissions are in effect, you can change them using the following format:

chmod who + (or −) permission file(s)<CR>

where:

chmod = Name of program,

who = One of three user groups *u*, *g*, *o*:
- *u* = User,
- *g* = Group, and
- *o* = Other.

+ − = Instruction that grants (+) or denies (−) permission.

permission = Authorization to *r*, *w*, or *x*:
r = Read,
w = Write, and
x = Execute.

file(s) = File (or directory) name(s) listed; assumed to be branches from your working directory, unless you use full path (names).

This may sound a bit confusing. But, a few examples on how to use the **chmod** command should help to make permission possibilities clear.

Let's use the permissions for the file *display* to experiment with **chmod**. You can see from the permissions that as the user and owner of *display* you can read, write, and run this executable file. You can protect the file against accidentally changing it by denying yourself write (*w*) permission by typing the command line **chmod u−w display<CR>**. After receiving the **$** prompt, type in **ls −l<CR>** to verify the permission has changed.

```
$ chmod u-w display<CR>
$ ls -l<CR>
total 35
-r-xr-xr-x   1  starship project   9346  Nov 1  08:06  display
-rwx--x--x   1  starship project   6428  Dec 2  10:24  list
drwx--x--x   2  starship project     32  Nov 8  15:32  tools
$
```

From this output, you can see that you no longer have permission to write changes into the file, that is, unless you change the mode back to include write permission.

Now, let's consider another example. Notice that permission to write into the file *display* has been denied to members of your group and other system users. These users, however, have read permission, which means that any of these users can copy the file into their own directories and then make changes to it. To prevent all system users

from copying this file, you could deny them read permission by typing **chmod go−r display<CR>**. The *g* and *o* stand for group members and all other system users, respectively, and the **−r** denies them permission to read or copy the file. Check the results with the **ls −l** command.

```
$ chmod go−r display<CR>
$ ls −l<CR>
total 35
-rwx--x--x  1  starship project   9346  Nov 1  08:06  display
-rwx--x--x  1  starship project   6428  Dec 2  10:24  list
drwx--x--x  2  starship project     32  Nov 8  15:32  tools
$
```

A Note on Permissions and Directories. If you read the preceding pages describing the **chmod** command, you might have gathered that you can use this command to grant or deny permission for directories as well as files. It is true, you can. To do so, simply use the directory name instead of a file name on the command line.

The impact, however, of granting or denying permissions for directories to various system users is worth considering. For example, if you grant read permission for a directory to yourself (u), members of your group (*g*), and other system users (*o*), every user who has access to the system can read the names of the files that directory contains by using the **ls −l** command. Similarly, granting write permission allows the designated users to create new files in the directory and change and remove existing ones. And granting permission to execute the directory allows the designated users the ability to move to that directory (and make it their working directory) by using the **cd** command.

An Alternate Method. The **chmod** method described in the preceding pages is one of two ways to change permissions to read, write, and execute files and directories. The method previously described uses symbols, such as *r, w, x* and *u, g, o,* to specify instructions to **chmod**. Hence, it is called the symbolic method.

The alternate method uses a number system called octal that is different than the decimal number system we typically use on a day-to-day basis. This method uses three octal numbers ranging from 0 through 7 to assign permissions. If you wish to use the octal method when changing permission, see the description of **chmod** in the *UNIX System User Reference Manual.*

Summary. The command recap that follows provides a quick reference on how **chmod** works.

Command Recap

chmod - change permission modes for files (and directories)

command	*instruction*	*arguments*
chmod	**who + − permission**	**filename(s)** **directoryname(s)**

Description: **chmod** gives (+) or removes (−) *read, write,* and *execute* permissions for three types of system users: *user* (you), *group* (members of your group), and *other* (all other users able to access the system on which you are working).

Remarks: The instruction set can be represented in either octal or symbolic terms.

Advanced Commands

You will become more and more familiar with the file system as you use the commands thus far discussed in this chapter. As this familiarity increases so might your need or interest for more

sophisticated information processing techniques when working with files. This section introduces you to three commands that give you just that. These commands and their capabilities are listed as follows:

diff -- Finds difference between two files,

grep -- Searches a file for a pattern, and

sort -- Sorts and merges files.

The following discussion only scratches the surface on information processing techniques available with the UNIX system. You may refer to the *UNIX System User Reference Manual* for additional information.

Identifying Differences Between Files (diff)

The **diff** command locates all the differences between two files and proceeds to tell you how to change the first file to be a carbon copy of the second. It reports all differences between the files.

The basic format for the command is:

diff file1 file2<CR>

If *file1* and *file2* are identical, the system returns the **$** prompt to you. If not, the **diff** command instructs you on how to bring the first file into agreement with the second by using line editor (**ed**) commands. (See *Chapter 5* for details on the line editor.) The UNIX system flags lines in *file1* with the < symbol and *file2* with the > symbol.

For example, if you use the **diff** command to identify differences between the files *johnson* and *sanders*, the system would respond as follows:

```
$ diff johnson sanders<CR>
2,3c2,4
< to Mr. Johnson on the topic of
< office automation.
---
> to Mrs. Sanders inviting her to
> speak at your departmental
> meeting.
$
```

The first line of the system response is

2,3c2,4

which means lines 2 through 3 in the file *johnson* must be changed (designated by *c*) to lines 2 through 4 in the file *sanders*. The system then displays lines 2 through 3 in the file *johnson* as follows:

< *to Mr. Johnson on the topic of*
< *office automation.*

and lines 2 through 4 in the file *sanders*

> *to Mrs. Sanders inviting her to*
> *speak at our departmental*
> *meeting.*

If you make these changes (using the **ed** or the **vi** text editing program), the file *johnson* will be identical to the file *sanders*. Remember, the **diff** command tells you exactly what the differences are between the named files. If you simply want an identical copy of a file, use the **cp** command.

Refer to the recap that follows for a summary of what you can expect the **diff** command to do when no options are specified. See the reference to the *UNIX System User Reference Manual* for details on available options.

Command Recap

diff - finds differences between two files

command	*options*	*arguments*
diff	available*	**file1 file2**

Description: **diff** reports what lines are different in two files and what you must do to make the first file identical with the second.

Remarks: Instructions on how to change a file to bring it into agreement with another file are line editor (**ed**) commands: *a* (append), *c* (change), or *d* (delete). Numbers given with *a*, *c*, or *d* indicate the lines to be modified. Also used are the symbols < (indicating a line from the first file) and > (indicating a line from the second file).

* See the *UNIX System User Reference Manual* for all available options and an explanation of their capabilities.

Searching a File for a Pattern *(grep)*

You can request the UNIX system to search through files for a specific word, phrase, or group of characters by using the **grep** command. Technically, **grep** means **g**lobally search through a file or files to locate a **r**egular **e**xpression and **p**rint the lines that contain the regular expression. Put simply, a regular expression is the pattern of characters--be it a word, a phrase, or an equation--that you stipulate.

The basic format for the command line is:

grep pattern file(s)<CR>

Thus, to locate the line containing the word *automation* in the file *johnson*, you would type:

grep automation johnson<CR>

and the system would respond as follows:

```
$ grep automation johnson<CR>
office automation
$
```

The output gives you all the lines in the file *johnson* that contain the pattern for which you were searching, which is the word *automation*.

If the pattern contains multiple words or any characters that have a special meaning to the UNIX system, such as **$**, |, *, **?**, and so on, the entire pattern must be enclosed in single quotes. (For an explanation of the special meaning for these and other characters see the section entitled *Metacharacters* in *Chapter 7, Shell Tutorial.*) For example, if you are interested in locating the lines containing the pattern *office automation*, the command line and system response would read:

```
$ grep `office automation' johnson<CR>
office automation.
$
```

But what if you could not recall to whom you sent a letter on the topic of office automation in the first place--Mr. Johnson or Mrs. Sanders? You could type:

grep `office automation' johnson sanders<CR>

If you did, the system would respond in the following manner:

```
$ grep `office automation' johnson sanders<CR>
johnson:office automation.
$
```

The output tells you that the pattern *office automation* is found once in the file *johnson*.

In addition to the capabilities of the **grep** command that are summarized in the recap that follows, the UNIX system provides variations to the basic **grep** command, called **egrep** and **fgrep**, along with several options that further enhance the searching powers of the command. See the *UNIX System User Reference Manual* if you are interested in learning more.

Command Recap

grep - searches a file for a pattern

command	*options*	*arguments*
grep	available*	**pattern file(s)**

Description: **grep** searches the file or files you name for lines containing a pattern and then prints the lines that match. If you name more than one file, the name of the file containing the pattern is given also.

Remarks: If the pattern you give contains multiple words or special characters, enclose the pattern in single quotes on the command line.

* See the *UNIX System User Reference Manual* for all available options and an explanation of their capabilities.

Sorting and Merging Files (sort)

The UNIX system provides you with an efficient tool called **sort** for sorting and merging files. The basic form of the command line is:

sort file(s)<CR>

which causes lines in the specified files to be sorted and merged in the order defined by the ASCII representations of the characters in the lines.

- Lines beginning with numbers are sorted by digit and listed before letters in the output,
- Lines beginning with uppercase letters are listed before lines beginning with lowercase letters, and
- Lines beginning with symbols, such as *, %, or @, are sorted on the basis of the symbol's ASCII representation.

To get an idea of how the **sort** command works, let's say that you have two files, named *phase1* and *phase2*, each containing a list of names that you wish to sort alphabetically and finally interfile into one list. First, display the contents of each file using the **cat** command.

```
$ cat phase1<CR>
Smith, Allyn
Jones, Barbara
Cook, Karen
Moore, Peter
Wolf, Robert
$ cat phase2<CR>
Frank, M. Jay
Nelson, James
West, Donna
Hill, Charles
Morgan, Kristine
$
```

(Note: we could have used the command line **cat phase1 phase2<CR>** instead of listing the contents of each file separately.)

Now, sort and merge the contents of the two files using the **sort** command. Note that the output of the **sort** program will print on the terminal monitor unless you specify otherwise.

```
$ sort phase1 phase2<CR>
Cook, Karen
Frank, M. Jay
Hill, Charles
Jones, Barbara
Moore, Peter
Morgan, Kristine
Nelson, James
Smith, Allyn
West, Donna
Wolf, Robert
$
```

In addition to putting together simple listings as in the previous examples, the **sort** command can rearrange the lines and parts of lines (called fields) according to a number of other specifications you can designate on the command line. The possible specifications are complex and are not within the scope of this text. You should consult the *UNIX System User Reference Manual* for a full rundown on the available options.

However, the following command recap summarizes the capabilities of the **sort** program.

Command Recap

sort - sorts and merges files

command	*options*	*arguments*
sort	available*	**file(s)**

Description: **sort** sorts and merges lines from the file or files you name and displays the result on your terminal.

Remarks: If no options are specified on the command line, lines are sorted and merged in the order defined by the ASCII representations of the characters in the lines.

* See the *UNIX System User Reference Manual* for all available options and an explanation of their capabilities.

SUMMARY

This chapter described the structure of the file system and presented ways to use and to navigate through the file system via UNIX system commands. The next chapter gives you an overview of a variety of UNIX system capabilities, such as text editing, using the shell as a command language, communicating electronically with other system users, and programming and developing software.

Chapter 4

UNIX SYSTEM CAPABILITIES

PAGE

INTRODUCTION 4-1

TEXT EDITING 4-1
- What Is a Text Editor? 4-2
- How Does a Text Editor Work? 4-2
  - Text Editing Buffers 4-2
  - Modes of Operation 4-3
- Line Editor 4-4
- Screen Editor 4-5

WORKING IN THE SHELL 4-6
- Using Shell Shorthand 4-7
- Redirecting the Flow of Input and Output 4-9
  - Redirecting the Standard Output *(>)* 4-11
  - Redirecting and Appending the Standard Output *(>>)* 4-13
  - Redirecting the Standard Input *(<)* 4-13
  - Connecting Commands with the Pipe *(|)* 4-14
  - Summary 4-16
- Running Multiple Programs 4-16
  - Executing Commands in Sequence 4-16
  - Executing Commands Simultaneously 4-17
- Customizing Your Computing Environment 4-19

COMMUNICATING ELECTRONICALLY 4-20

PROGRAMMING IN THE SYSTEM 4-21
- Programming in the Shell 4-21
- Programming in the C Language 4-23
- Other Programming Languages 4-24
- Tools to Aid Software Development 4-25
  - Source Code Control System *(SCCS)* 4-25
  - Remote Job Entry *(RJE)* 4-26
  - Maintaining Programs *(make)* 4-26
  - Generating Programs for Lexical Tasks *(lex)* 4-27
  - Generating Parser Programs *(yacc)* 4-27

Chapter 4

UNIX SYSTEM CAPABILITIES

INTRODUCTION

This chapter serves as a transition between the first three chapters in the overview part of this guide and the four tutorials that follow. The material in this chapter combines basic, fundamental concepts about the UNIX system covered in the first three chapters of this guide with information about system capabilities that you may use to do your computing work efficiently and effectively.

This chapter provides an overview of the following UNIX system capabilities: text editing, working in the shell, communicating electronically, and programming in the UNIX system environment. In addition, it serves as an introduction to chapters 5, 6, 7, and 8--*Line Editor Tutorial, Screen Editor Tutorial, Shell Tutorial,* and *Communication Tutorial,* respectively.

TEXT EDITING

You have read a good deal about files up to this point simply because using the file system is a way of life in a UNIX system environment. The information in this section will enhance your knowledge about manipulating files by introducing you to a software tool called a text editor. A text editor provides you with the ability to create and modify files: it will help you to fare well in the UNIX system since a considerable amount of your computing time may be spent writing and revising letters, memos, reports, or source code for programs that will be stored in files.

This section contains information that tells you what a text editor is and how it works. In addition, this section acquaints you with two types of text editors supported on the UNIX system: the line editor and the visual, or screen, editor. Since you will probably come to prefer one of these editing programs over the other--even if you learn to use them equally well--the line editor and the screen editor

are briefly compared to help you to assess their capabilities. For detailed information on the line editor and the screen editor, see *Chapter 5* and *Chapter 6*.

What Is a Text Editor?

When you write or type letters, memos, and reports and then decide to change what you have written or typed, you will use skills required in text editing. These skills include inserting new or additional material, deleting unneeded material, transposing material (sometimes called cutting and pasting), and finally preparing a clean, corrected copy. Text editors perform these tasks at your direction making writing and revising text much easier and quicker than if done by hand or on a typewriter.

In the UNIX system, a text editor is much like the UNIX system shell. Both a text editor and the shell are programs that accept your commands and then perform the requested functions--essentially, they are both interactive programs. A major difference between a text editor and the shell, however, is the set of commands that each recognizes. All the commands you have learned up to this point belong to the shell's command set. A text editor, on the other hand, has its own distinct set of commands that allow you to create, move, add, and delete text in files, as well as acquire text from other files.

How Does a Text Editor Work?

To understand how a text editor works you need information about the environment created when you use an editing program and the modes of operation understood by a text editor.

Text Editing Buffers

To create a new file, you must ask the shell to put the editor in control of your computing session. When you do, a temporary work space is allocated to you by the editor. This work space is called the editing buffer, in it you can enter information you want the file to hold and modify it if you wish.

Because you are in a temporary work space when using a text editor, the file you are creating along with the changes you make to it are also temporary. This work space allotment and what it is holding

will exist only as long as you work in the editing program. If you wish to save the file, you must tell the text editor to write the contents of the buffer into a storage area. If you do not tell the editor to write or record what you have done during the editing session, the buffer's contents will disappear when you leave the editing program. If you forget to write a new file or update an existing one, the text editors remind you to do so when you attempt to leave the editing environment.

To modify an existing file, the procedure is almost identical to the one you follow when creating a new file. First, call the editor and give it the name of the file you wish to change. In turn, the editor makes a copy of the file that is in the storage area and places it in the buffer so you can work on it.

When you finish editing the file, you can write the buffer's contents into storage and leave the editing program knowing the file is updated and ready to be recalled when you need it again. Or you can chose to leave the editor without writing the file if you have made a critical mistake or you are unhappy with the edited version. This step leaves the original file intact and the edited copy disappears.

Regardless of whether you are creating a new file or updating an existing one, the text you put in the buffer is organized into lines. A line of text is simply the series of characters that appears horizontally across a row of typing that is ended by pressing the **<CR>** key. Occasionally, files may contain a line of text that is too long to fit on the terminal monitor. Some terminals will automatically display the continuation of the line on the next row of the monitor, whereas others will not.

Modes of Operation

Text editors are capable of understanding two modes of operation: the command mode and the text input mode.

When you begin an editing session, you will automatically be placed in command mode. In command mode, all your input is interpreted as a command. Typical editing commands allow you to move about in a file, search for patterns in the file's contents, or print a portion of a file on the terminal monitor. The input mode is entered when you

use a command to create text. Once in input mode, what you type on the keyboard is placed into the buffer as part of the text file until you send the appropriate instruction to the editor that returns you to command mode.

You may occasionally lose track of the mode in which you are working by attempting to enter text while in command mode or by trying to enter a command while in input mode. This is something even experienced users do from time to time. It will not take long to recognize the mistake and it will be apparent what to do to remedy these situations as you work through the tutorials in *Chapter 5* and *Chapter 6*.

Line Editor

The line editor, accessed by the **ed** command, is a fast, versatile program for preparing text files. This editor gets its name because it operates on the lines of text a file holds. For example, to change a single character in a file, you specify the line of the file that contains the character you wish to change and then specify the change.

Put simply, you manipulate text on a line-by-line basis with the line editor. Commands for this text editor can change lines, print lines, read and write files, and initiate text entry. In addition, you can specify the line editor to run from a shell program; something you cannot do with the screen editor. (See *Chapter 7* for information on basic shell programming techniques.)

The line editor works equally well on paper printing terminals and video display terminals. It will also obligingly accommodate you if you are using a slow-speed telephone line.

Refer to *Chapter 5, Line Editor Tutorial,* for instructions on how to use this editing tool. Also see *Appendix D* for a summary of line editor commands. If you are interested in a comparison of line editor (**ed**) and screen editor (**vi**) features, see *Table 4-1*.

TABLE 4-1

Comparison of Line *(ed)* and Screen *(vi)* Editors

Feature	Line Editor *(ed)*	Screen Editor *(vi)*
Recommended terminal type	Paper-printing or VDT*	VDT
Speed	Accommodates high- and low-speed data transmission lines.	Works best via high-speed data transmission lines (1,200+ baud).
Versatility	Can be specified to run from shell scripts as well as used during editing sessions.	Must be used interactively during editing sessions.
Sophistication	Changes text quickly. Uses comparatively small amounts of processing time.	Changes text easily. However, can make heavy demands on computer resources.
Power	Provides a full set of editing commands. Standard UNIX system text editor.	Provides its own editing commands and recognizes all line editor commands as well.

* VDT = video display terminal

Screen Editor

The screen editor, accessed by the **vi** command, is a display-oriented, interactive software tool. When you use the screen editor, your terminal acts as a window to let you view the file you are editing a screenful or page at a time. This editor works most efficiently and effectively when used on a video display terminal operating at 1,200 or higher baud.

For the most part, modifications to a file (such as, additions, deletions, and changes) are accomplished by positioning the cursor at the point in the window where the modification is to be made and then making the change. In other words, the screen editor displays the effects of editing changes in the context in which you make them.

Because of this feature, the screen editor in considered to be much more sophisticated than the line editor.

Furthermore, the screen editor offers a replete collection of commands. For example, a number of screen editor commands allow you to move the cursor around within the window to a file. Other commands move the window up or down through a page or more of the file. Still other commands allow you to change existing text or to create new text. In addition to its own set of commands, the screen editor has access to all the commands offered by the line editor. This arsenal of commands accounts for the screen editor's tremendous power.

There is, however, a trade-off for the screen editor's speed, visual appeal, efficiency, and power, which is the heavy demand that it places on the computer's processing time. For example, a simple change might cause an entire screen to need updating. Moreover, if simple changes lead to long delays while you wait for a screen to be updated, the pleasant experience of using a visual-oriented editor can be somewhat diminished.

Refer to *Chapter 6, Screen Editor Tutorial,* for instructions on how to use this software. And see *Appendix E,* which contains a summary of screen editor commands. If you wish to compare the features of the line editor (**ed**) and the screen editor (**vi**) see *Table 4-1.*

WORKING IN THE SHELL

Every time you log into the UNIX system you will be communicating directly with a program called the shell. You will continue to interact with the shell until you log off the system, unless you use a program, such as a text editor, that temporarily suspends your dealings with the shell until you are finished using that particular program.

The shell is much like other programs, except that instead of performing one job, as **cat** or **ls** does, it is central to most of your interactions with the UNIX system. This is because the shell's primary function is to act as an interpreter between you and the computer on which the UNIX system is running. As an interpreter,

the shell translates your requests into language the computer understands, calls requested programs into memory, and executes them.

This section acquaints you with some of the ways you can use the shell as the command language interpreter to simplify a computing session and to enhance your ability to use system features. In addition to running a single program for you, you can also use the shell to:

- interpret the name of a file or a directory you input in an abbreviated way using a type of "shell shorthand,"
- redirect the flow of input and output of the programs you run,
- execute multiple programs, and
- tailor your computing environment to meet your individual needs and preferences.

In addition to being the command language interpreter, the shell is also a programming language. If you would like an overview of shell programming capabilities, see the section entitled *Programming in the System* at the end of this chapter. Or refer to *Chapter 7, Shell Tutorial,* for detailed information on how to use the shell as a command language interpreter and as a programming language. A separate document, *UNIX System Shell Commands and Programming,* should be consulted for complete, unabridged information on shell programming.

Using Shell Shorthand

Many UNIX system commands require that you name a file or a directory as an argument to it on a command line, such as **mkdir directory name(s)<CR>** or **rm filename(s)<CR>**. Easy enough! But suppose you have 12 files to remove corresponding to monthly reports for 1983 named *rept1, rept2, rept3, rept4,* and so on? Or suppose you need to move 24 files corresponding to file names *sect1, sect2, ... sect24* to a different directory?

Typing the file name for each monthly report after the **rm** command or the file name for each section after the **mv** command is still easy, but all the repetition gets tedious after inputting four or five names.

In instances like these, you should consider using shorthand notation when specifying file or directory names. If the file or directory names have some part in common, you can use a type of shorthand to tell the shell that you are referring to all of them on the basis of the similarity without specifying each one individually. Or, if a file has a unique character or sequence of characters within a group of similarly named files, you can use this shorthand notation to locate the file on the basis of the difference.

The UNIX system recognizes several characters as having special meanings when they are used in place of a directory name or when they appear as part of a file or directory name on a command line. These characters allow you to specify the names of files and directories in a rapid, abbreviated way. Some of the characters are referred to as metacharacters because of their special meanings to the shell.

The special characters are **. .. ? * [ ]** – \ and their meanings are summarized in *Table 4-2*. When you specify file or directory names, you can substitute various characters within them with the appropriate shorthand abbreviation. Any part of the name that is not a special character is taken at its literal value.

For example, for the possibilities described at the beginning of this section, typing **rm rept*<CR>** would remove all the files in the current directory starting with the characters *rept* followed by any other characters corresponding to monthly reports for 1983, and typing **mv sect* ../chapter3<CR>** would move all the files from the current directory beginning with the letters *sect* and followed by any other characters to another directory named *chapter3* belonging to its parent directory.

Details on how to use the special characters appear in other chapters of this guide as indicated in *Table 4-2*. Refer to that chapter for the information you need.

TABLE 4-2

Shorthand Notation for File and Directory Names

Special Character	Meaning	Detailed Reference
.	Current directory	*Chapter 3*
..	Parent directory	*Chapter 3*
?	Match any single character	*Chapter 7*
*	Match any number of characters	*Chapter 7*
[ ]	Designate a sequence of characters to be matched, such as [abc] or [628]	*Chapter 7*
–	Specify a character range within [ ], such as A-Z	*Chapter 7*
\	Remove meaning of special characters	*Chapters 2, 7*

Redirecting the Flow of Input and Output

Up to this point in the *UNIX System User Guide*, any request to ask the shell to execute a command was done by inputting the command and the necessary argument(s) on the terminal keyboard. In turn, the output, if any, was displayed on the terminal monitor. This pattern illustrates the idea of standard input and standard output.

In general, the place from which a program expects to receive its input is called the standard input. A UNIX system command called **mail**, which you will learn more about in *Chapter 8*, provides a good example of this and warrants mentioning here. For example, to use **mail**, you would simply type **mail jmrs<CR>** and the **mail** command takes everything you type on your keyboard after **<CR>** until you type **<^d>** as input. After you type **<^d>**, mail sends your input to the person with the login name *jmrs*. The place to which a program writes its results, in this case the login name *jmrs*, is referred to as the standard output.

In the UNIX system, most commands expect to receive their input from the keyboard and then display output on the terminal monitor.

By default, then, the standard input is the keyboard and the standard output is the terminal monitor (*Figure 4-1*).

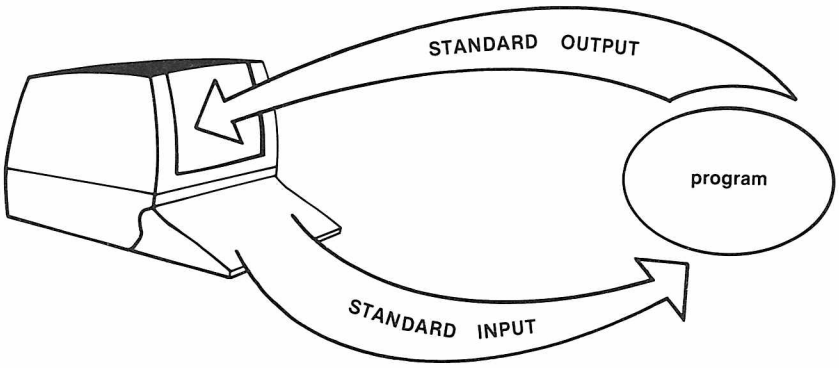


Figure 4-1. Standard input/output flow. A program's standard input and standard output are usually assigned to your terminal.

You can, if you wish, use a feature called redirection to change these defaults. Put simply, redirection is a UNIX system feature that allows you to request the shell to reassign standard input and/or standard output to other files or devices.

With the redirection feature, you can request the shell to do the following:

- reassign to a file any output that a program would ordinarily send to your terminal,
- have a program take its input from a file rather than from your terminal keyboard, or
- use a program as the source of data for another program.

You request the shell to redirect input and output using a set of operators, which are > (greater than sign), >> (two greater than signs), < (less than sign), and | (a pipe). Now let's take a look at what each of these operators can do for you.

Redirecting the Standard Output (>)

The > operator allows you to redirect the output of a command (or program) into a file (*Figure 4-2*).

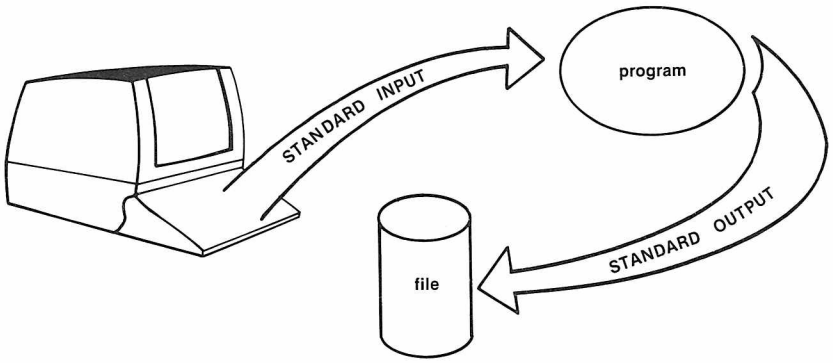


Figure 4-2. Standard output can be redirected from your terminal to a file.

To use the > operator, follow the command line format:

command > newfile<CR>

in which you can choose to surround the > operator with spaces as indicated in the command line or leave the spaces out (**command>newfile<CR>**); either method is correct.

For example, if you have two files, named *group1* and *group2* each containing a list of names with telephone extension numbers that you would like to sort alphabetically and then interfile into a separate file called *members*, you would type:

sort group1 group2 > members<CR>

When you do, the UNIX system first alphabetically sorts and then interfiles the contents of the files *group1* and *group2* and redirects the

output into the file called *members* rather than displaying it on your terminal. If you wish to read the contents of the *members* file, you could use the **cat** or **pg** command.

Therefore, if the contents of the file *group1* is:

Smith, Allyn	101
Jones, Barbara	203
Cook, Karen	521
Moore, Peter	180
Wolf, Robert	125

and the contents of the file *group2* is:

Frank, M. Jay	118
Nelson, James	210
West, Donna	333
Hill, Charles	256
Morgan, Kristine	175

then the file *members* would appear as follows on your terminal when displayed with the **cat** command.

```
$ sort phase1 phase2 > members<CR>
$ cat members<CR>
Cook, Karen          521
Frank, M. Jay        118
Hill, Charles        256
Jones, Barbara       203
Moore, Peter         180
Morgan, Kristine     175
Nelson, James        210
Smith, Allyn         101
West, Donna          333
Wolf, Robert         125
$
```

Keep in mind that if the file to which you are redirecting the standard output already exists, its contents will be replaced with the output of the redirection command.

Redirecting and Appending the Standard Output (>>)

Occasionally, you might like to add information to the end of an existing file. You can use the >> operator to do so. Simply input the following command line:

command >> file<CR>

For example, if the file *members* that was created in the previous section was subject to additions and deletions, it might be a good idea to date the list so you know at a glance what version of the list you are using. You could do so by typing

date >> members<CR>

on the command line and the date and time would be printed at the end of the file *members*. Or instead of adding the date to the end of the file *members*, you could have appended another file containing even more names.

Redirecting the Standard Input (<)

Standard input can be redirected as well as standard output with the < operator. The general command line format for input redirection is:

command < file<CR>

in which the < operator informs the command (or program) to take input from the file you specify rather than from the terminal keyboard (*Figure 4-3*).

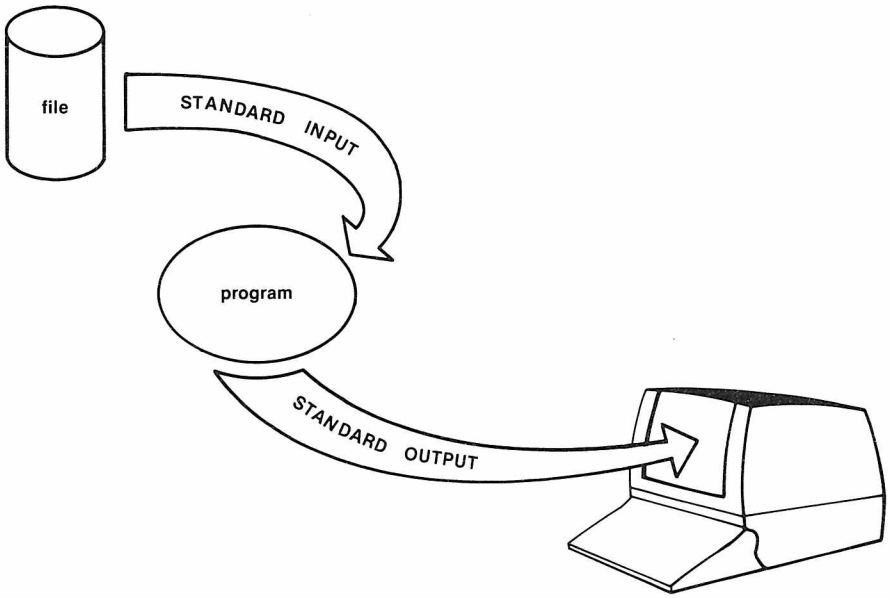


Figure 4-3. You can ask the shell to take a program's input from a file rather than from your terminal.

For example, if you would like to send a copy of the file *members* to co-workers who work on your UNIX system and who have the login names *mary2* and *jmrs*, typing

mail mary2 jmrs < members<CR>

will accomplish the task. The **mail** command, however, does not know whether it received its input from the file *members* (which it did) or from your keyboard. Rather, input/output redirection is a service provided by the UNIX system shell and is available to every program. (You will learn more about the **mail** command in *Chapter 8*.)

Connecting Commands with the Pipe (|)

The pipe operator is a powerful, yet flexible, mechanism for doing computing tasks quickly and without the need to develop special

purpose tools. You can use it to redirect the standard output of one program to be the standard input of another (*Figure 4-4*). Generally, the format for using the pipe is:

command | command<CR>

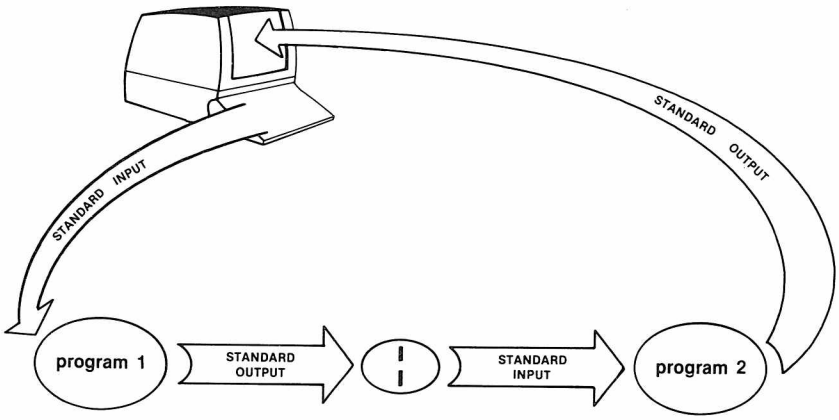


Figure 4-4. You can use the output from one program to be the input for another.

A popular example of this is taking the output of the **who** command (which you were introduced to in *Chapter 2*) and using it as input to the **wc** command (which counts lines, words, and/or characters) as follows:

who | wc -l<CR>

This example shows that the standard output of the **who** command was passed to the **wc -l** command (**-l** is the option that counts the number of lines output by the **who** command, each corresponding to a user who is logged into your UNIX system.)

Summary

Table 4-3 summarizes which operator performs which redirection task and what general format should be followed in using it. Refer to the section on redirection in *Chapter 7* for details on how to use them.

TABLE 4-3

Options for Redirecting Input and/or Output†

Action	Operator	General Format
Redirecting output to a file	>	command > filename
Redirecting and appending output to a file	>>	command >> filename
Redirecting input from a file	<	command < filename
Redirecting output of first command to be input for second	|	command | command

* See *Chapter 7 for complete details on how to use these options.*

† *Blank spaces immediately before and after redirection operators are optional.*

Running Multiple Programs

There are two methods for running multiple programs: you can specify more than one command to execute in sequence from a single command line or you can run commands simultaneously.

Executing Commands in Sequence

Up to this point, the command lines to which you have been introduced and examples for using them have dealt with asking the shell to run a single request or program. For example, each of the command lines **cat filename<CR>**, **date<CR>**, and **ls -l directoryname<CR>** requests the shell to perform one task. You can, however, ask the shell to execute more than one request per command line. Sequential execution allows you to enter as many commands as you wish on one command line and have them execute in the order in which you input them.

To do so, you should first be familiar with the general rules for command line syntax given in *Chapter 3*. Briefly, command line syntax orders elements in the command line so that the command name, any options you wish to specify, and the data on which the command is to operate (usually the name of a file or directory) follow one another.

To execute more than one command on a line, simply separate the request sequences with semicolons (;) as follows:

command option(s) argument(s); command option(s) argument(s); ...<CR>

For example, to determine where you are in the file system and then list the contents of the directory in which you are working, you can type **pwd; ls<CR>** and the terminal monitor might read:

```
$pwd; ls<CR>
/user1/starship/bin
dir
list
tools
$
```

As you can see, the output of the multiple commands is ordered the same way the input is: first, the current working directory is given (in response to **pwd**) and, then, the names of the files and/or directories it holds follow (in response to **ls**).

You could just as easily type **who am i; date; who<CR>** or **mkdir directoryabc; cd directoryabc; pwd<CR>** or any combination of commands that you wish to use.

Executing Commands Simultaneously

In addition to running programs sequentially, you can choose to run them simultaneously. To do so, you need to know the difference between foreground and background commands. When a program runs in the background, the computer is executing that program concurrently with the commands that you enter or with the program

that you run in the foreground. However, the computer considers your foreground work more important, in a sense, than your background program. This difference has no perceivable effect on the execution of most programs, but running a job in the background is a useful technique when you wish to run a lengthy or time-consuming job without tying up your terminal.

All the command lines used in this guide until now have been examples of foreground commands. This means that they were initiated and run to completion before other commands could be executed and before the shell would return the **$** prompt for you to continue. However, you also have the option of running a command in the background so you can continue to work in the foreground.

You can run a command in the background by placing an ampersand (**&**) at the end of the command line as follows:

command option(s) argument(s) &<CR>

When the shell reads the **&**, it starts running the program, prints an identification number, and displays the **$** prompt so you can use the terminal immediately for other work.

To save the output from the job you are running in the background, you must redirect the results of the execution into another file so you can look at or use the output when you are ready. For example, if you input the command **cat file1 file2 > file3 &<CR>**, the shell would first give you an identification number, and then the prompt. It will also save the results of **cat file1 file2** in a file named *file3*. When you are ready to peruse *file3*, simply use **cat** or **pg**. If you do not redirect the output, then no output is saved.

When a program is running in the background, it ignores interrupt and break signals, but if you log off, the shell terminates the background program along with the computing session. If you would like to stop a background command while you are still logged into the UNIX system, type **kill id<CR>**, where id is the identification number of the command. On the other hand, to have a program continue to run after you log off, you can use the **nohup** command (which stands for "no hang up") in the following way

nohup command &<CR>

When you do, the command will continue to run until completion and its output is saved in a file called *nohup.out* (which stands for **nohup** output).

Customizing Your Computing Environment

The information in this section deals with another dimension of control provided to you by the shell called your environment. When you log into the UNIX system, the shell automatically sets up a computing environment for you. You can choose to use it as supplied by the system or you can tailor it to meet your needs.

By default, the environment set up by the shell includes the variables:

HOME = your login directory,

PATH = route the shell takes to search for executable files or commands (typically *PATH=:/bin:/usr/bin*), and

LOGNAME = your login name.

If you find the default environment satisfactory, simply leave it as it is and go on with your work. However, if you would like to modify it, you must have a file in your login directory named *.profile*. If you do not, you can create one with a text editor like **ed** or **vi**.

To determine if you have a *.profile*, move to your login directory and type **cat .profile<CR>** and its contents should appear on the terminal monitor. Typically, the *.profile* tests for mail and sets data parameters, system variables, and terminal settings.

Possible modifications to your login environment might include changing your login prompt, setting tab stops, and changing erase and kill characters. If you would like to customize your *.profile*, see the section entitled *Modifying Your Login Environment*, in *Chapter 7*.

COMMUNICATING ELECTRONICALLY

Before the days of office automation, you would probably have thought of relaying a message or information to someone either personally or by way of a letter, note, or telephone conversation. Now as a UNIX system user, you can choose to communicate electronically with other UNIX system users by way of the computer.

You can send messages or transmit information stored in files to other users who work on your system or on another UNIX system. To do so, your UNIX system must be able to communicate with the UNIX system to which you wish to send information. In addition, the command you use to send information depends on what you are sending.

This guide introduces you to these communication programs:

mail -- This command is typically used for sending messages to others and reading the messages sent to you. You can use **mail** to send messages or files to other UNIX system users using their login names as addresses. And, at your convenience, you can use the **mail** command to read messages sent to you by other users. With **mail**, the recipient can choose when to read it.

uuto/uupick -- These commands are used to send and retrieve files. You use the **uuto** command to send a file(s) to a public directory; when its available to the recipient, the person is sent mail telling him/her that the file(s) has arrived. The recipient then can use the **uupick** command to copy the file(s) from the public directory to the directory of choice.

mailx -- This command is a sophisticated, more powerful spin-off of **mail**. It offers a number of options for managing the electronic mail you send and receive.

Chapter 8 teaches you how to use the **mail**, **uuto**, and **uupick** commands. It also introduces you to the **mailx** command so you can begin to use it.

PROGRAMMING IN THE SYSTEM

The UNIX system provides an efficient, effective, and convenient environment for programming and software development. This section briefly describes the environment and your programming options when working in it.

If you are not a programmer, your immediate reaction might be to skip this section. But you need not be a programmer or software developer to enjoy some of the capabilities that fall under the realm of programming.

For example, you can use the shell as a command level programming language as well as the command line interpreter. Shell programming capabilities are useful and usable techniques that allow you to take simple, existing programs and make them more powerful. So why not read on.

On the other hand, if you're interested in sophisticated programming and software development capabilities, this section can serve as a springboard to using them.

What you can expect to find in the next few pages is an overview of shell and C language programming and a mention of other languages that can be used on the UNIX system. In addition, an overview of some UNIX system tools for software development is included.

Programming in the Shell

Most interactive users of the UNIX system think of the shell solely as the command language interpreter. The shell, however, is also a command level programming language. What this means is that you can let the shell continue to act as your liaison with the computer or you can program the shell to repeat sequences of instructions and to test certain considerations for you automatically. When you program the shell to perform a task, you use the shell to read and to execute commands that you place in an executable file. These files are sometimes called shell scripts or shell procedures.

When you use the shell in this manner, it provides you with features, like variables, control structures, subroutines, and parameter passing

that are very similar to those offered by programming languages. These features provide you with the ability to create your own tools by linking together system commands.

For example, you can write a simple shell procedure from existing UNIX system programs that tells you the date and time along with the number of users working on your UNIX system. One way to do so is illustrated in the following screen:

```
$ cat > users<CR>
date; who | wc -l<CR>
<^d>
$ chmod u+x users<CR>
$
```

A file called *users* is created using the > redirection operator. In the example, **cat** is taking as input everything you type after **<CR>** on the command line and placing it in a file named *users*. Then the file is made executable with the **chmod** command. If you type the command **users<CR>**, your terminal monitor would look something like the next screen.

```
$ users<CR>
Tues May 22 10:29:09 CDT 1984
   7
$
```

The output tells you that seven users were logged into the system when you typed the command at approximately 10:30 A.M. on Tuesday, May 22.

For additional information on shell procedures and for more sophisticated shell programming techniques, see *Chapter 7, Shell Tutorial,* for step-by-step instructions.

Programming in the C Language

C is a general purpose programming language. It is a relatively "low level" language, which means that C deals with the same sort of objects that most computers do, namely characters, numbers, and addresses. These may be combined and moved about with the usual arithmetic and logic operators.

C is closely associated with the UNIX system because it was developed on the UNIX system and because UNIX system software is largely written in C.

Although the C programming language is implemented on many computers, it is independent of any particular machine architecture. With a little care, it is easy to write portable programs, that is, programs that can be run without change on a variety of computers if the machine supports a C compiler.

The C programming language comprises the following main elements:

- *Types, operators, and expressions*--Constants and variables are the basic data objects manipulated in a program. Constants are data objects that do not change during the execution of a program, while variables are assigned new values throughout execution. Declarations list variables, state type, and perhaps initial values. Operators specify what is to be done on them. Expressions combine variables and constants to produce new values.

- *Control flow*--Control flow statements of a language specify the order in which computations are done. In C, these include *if-else, else-if,* and *switch* statements, and *while, for,* and *do-while* loops. In addition, *break, continue,* and *goto* statements can be used. Labels can be used as well.

- *Functions and program structure*--C programs generally consist of numerous small functions rather than a few big ones. These functions break large computing tasks into smaller ones and enable you to build on what others have done.

- *Pointers and arrays*--A pointer is a variable that contains the address of another variable. Pointers are frequently used when programming in C because oftentimes they provide the only way to express a computation and partly because their use typically leads to more compact and efficient code than can be obtained in other ways.

- *Structures*--A structure is a collection of one or more variables, possibly of different types, that are grouped together under a single name for convenient handling. Structures help to organize complicated data because they permit a group of related variables to be treated as a unit instead of separate entities.

- *Input and output*--A standard I/O library containing a set of functions designed to provide a standard input and output system is available for C programs. This library is a UNIX system feature available for programming in C.

These elements are covered in detail in *The C Programming Language* by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). Additional information is also available in the *UNIX System Programming Guide*.

Other Programming Languages

In addition to C, other programming languages are available for use on the UNIX system, such as FORTRAN-77, BASIC, Pascal, COBOL, APL, LISP, and SNOBOL.

You can obtain details on FORTRAN and its variations in the *UNIX System Programming Guide*. Or contact your AT&T Technologies Account Representative for document availability and ordering information on the others.

Tools to Aid Software Development

This section highlights some sophisticated software development tools available on the UNIX system. The tools are designed to make development of software easier and to provide you with a systematic approach to programming.

There are numerous software development aids provided by the UNIX operating system. This section introduces you to five of them to give you an idea of what you can expect development utilities to do. They are:

SCCS -- Source Code Control System,

RJE -- Remote job entry,

make -- Maintaining programs,

lex -- Generating programs for simple lexical tasks, and

yacc -- Generating parser programs.

Refer to the *UNIX System Support Tools Guide* and the *UNIX System Programming Guide* for more information.

Source Code Control System *(SCCS)*

The Source Code Control System (SCCS) is a collection of UNIX system commands that helps you to control and report changes to source code files or text files. SCCS allows you to access different versions of the same file while maintaining only one file. The way this works is that SCCS stores the original file on a disk. Whenever modifications are made to the file SCCS stores only those changes as a set in something called a *delta*. Each delta or set of changes is numbered to reflect the different versions of a file. You can then choose to retrieve either the original file or a version of the original file.

By allowing SCCS to store and control all iterations of a file, space allocations for storage are minimized and administration of different versions of the same program or document is efficient and simplified. Updates to files can be made quickly and the original version of a program or document is retained if you should need to recall it later.

For additional information, see the *UNIX System Support Tools Guide*. Most of the commands needed to use SCCS are documented in the *UNIX System User Reference Manual*.

Remote Job Entry *(RJE)*

Remote job entry (RJE) is a software package designed to facilitate communication between a UNIX operating system and an IBM System/360 or an IBM System/370 computer. The RJE software allows the UNIX operating system to communicate with the IBM Job Entry Subsystem by mimicking an IBM System/360 remote multileaving work station. A set of background processes support RJE, and the UNIX system uses these processes to submit jobs for remote execution on the networked IBM system.

When RJE software runs, it does so in the background. It transmits jobs (consisting of job control statements [JCL] and input data) that you queue with the **send** command and status reports you request with the **rjestat** command. In turn, the RJE software subsystem receives print and punch data sets and message output from the IBM system.

For more information on RJE software, see the *UNIX System Support Tools Guide*. Commands to be used with RJE are covered in the *UNIX System User Reference Manual* and the *UNIX System Administrator Reference Manual*.

Maintaining Programs *(make)*

The **make** command is a tool for maintaining, supporting, and regenerating large programs or documents on the basis of smaller ones. Since it is easier to handle and modify small programs, it is recommended that if you wish to develop a large program, you start by creating a series of smaller ones that work together to produce the large one.

The **make** command provides you with a method to store all the information you need to assemble small programs or modules into a large, more sophisticated one. A file called a **makefile** holds the file names of the small programs, the steps necessary to generate the large program, and specifies the dependencies among the files.

When **make** executes the **makefile**, the date and time you last modified any of the small programs are checked and the operations needed to update them are performed in sequence. Then, **make** goes on to create the overall large program.

For details on the operation of **make**, see the *UNIX System Support Tools Guide*. Or, for a quick reference, see the entry for **make** in the *UNIX System User Reference Manual*.

Generating Programs for Lexical Tasks (*lex*)

The **lex** utility generates programs to be used in simple lexical analysis of text. Lexical analysis is done by evaluating a stream of characters and constructing the forms that are acceptable to the language. Proper forms are defined in the **lex** program and usable forms can be defined by **lex** defaults or by you. **Lex** produces a subroutine as output that must be compiled and combined with other programs to use the lexical analyzer.

The processing done by the **lex** command can be the first step in creating a compiler-type program. In addition, it can be useful as a preprocessing tool for many different software generation functions.

For additional information on the **lex** command, see the *UNIX System Support Tools Guide*. A brief description of how **lex** operates and an explanation of its options can be found in the *UNIX System User Reference Manual*.

Generating Parser Programs (*yacc*)

The **yacc** program, short for **y**et **a**nother **c**ompiler **c**ompiler, is primarily used in the generation of software compilers. Essentially, **yacc** is a utility for creating parser subroutines. The way this works is that first **yacc** uses specified syntax and produces source code for a parser subroutine. Then, the parser subroutine is compiled, and finally used with a program that calls it to parse input. In this way,

structure can be imposed on the input to a program and the desired language can be created from defined rules.

See the *UNIX System Support Tools Guide* for details on the **yacc** command. Or refer to the *UNIX System User Reference Manual* for some general guidelines on how to use it.

UNIX SYSTEM TUTORIALS

Contents

Chapter 5. Line Editor Tutorial

Chapter 6. Screen Editor Tutorial

Chapter 7. Shell Tutorial

Chapter 8. Communication Tutorial

Chapter 5

LINE EDITOR TUTORIAL (ed)

PAGE

INTRODUCING THE LINE EDITOR 5-1

HOW TO READ THIS TUTORIAL 5-2

GETTING STARTED 5-3
- How to Access ed 5-4
- How to Create Text 5-5
- How to Display a Line of Text 5-6
- How to Delete a Line of Text 5-8
- How to Move Up or Down a Line in the File 5-9
- How to Save the Buffer Contents in a File 5-10
- How to Quit the Editor 5-11

EXERCISE 1 5-13

GENERAL FORMAT OF ed COMMANDS 5-13

LINE ADDRESSING 5-14
- Number Line Addresses 5-15
- Special Symbols Addresses 5-16
  - Current Line Address Character 5-16
  - Last Line Address Character 5-17
  - Address for the First Line Through the Last Line 5-18
  - Address for the Current Line Through the Last Line 5-18
- Relative Addressing, Adding or Subtracting Lines from the Current Line 5-19
- Character String Addresses 5-21
- Specifying a Range of Lines 5-24
- Specifying a Global Search 5-26

EXERCISE 2 5-29

PAGE

DISPLAY LINES IN A FILE 5-30
Display Lines of Text 5-30
Display Lines of Text Preceded by the Line Address Number 5-31

CREATING TEXT 5-33
Appending Text 5-33
Inserting Text 5-36
Changing Text 5-37

EXERCISE 3 5-39

DELETING TEXT 5-41
Deleting Lines of Text 5-41
Undo the Last Command 5-43
Deleting Commands in the Text Input Mode 5-44
Deleting the Current Line 5-44
Deleting the Last Characters Typed 5-45

SUBSTITUTING TEXT 5-47
Substituting on the Current Line 5-49
Substituting on One Line 5-50
Substituting on a Range of Lines 5-50
Global Substitution 5-52

EXERCISE 4 5-54

SPECIAL CHARACTERS 5-56

EXERCISE 5 5-67

MOVING TEXT 5-69
Move Lines of Text 5-69
Copy Lines of Text 5-72
Joining Contiguous Lines 5-74
Write Lines of Text to a File 5-75
Read in the Contents of a File 5-77

EXERCISE 6 5-79

PAGE

OTHER USEFUL COMMANDS AND INFORMATION 5-79
- Help Commands 5-79
- Display Nonprinting Characters 5-82
- The Current File Name 5-84
- Escape to the Shell 5-86
- Recover From a System Interrupt 5-87
- Conclusion 5-87

EXERCISE 7 5-88

ANSWERS TO EXERCISES 5-90
- Exercise 1 5-90
- Exercise 2 5-91
- Exercise 3 5-93
- Exercise 4 5-96
- Exercise 5 5-97
- Exercise 6 5-100
- Exercise 7 5-101

Chapter 5

LINE EDITOR TUTORIAL (ed)

INTRODUCING THE LINE EDITOR

This tutorial is an introduction to the line editor, **ed**. The advantages of the line editor are speed and versatility. **ed** requires very little computer time to perform editing tasks. The line editor commands can be typed in by you at a terminal, or they can be used in a shell program. (See *Chapter 7, Shell Tutorial.*)

When you enter **ed**, you are placed in a temporary buffer. The buffer is like a piece of scratch paper for you to work on until you have finished creating or correcting your text in this scratch pad buffer. If you are creating a new file, you enter commands from your terminal that tell **ed** how to create or modify your text in this scratch pad buffer. If you are editing an existing file, a copy of that file is placed in the buffer. Changes are made to the copy of the file. The changes have no effect on the original file until you instruct **ed**, using the "write command", to move the contents of the scratch pad buffer into the file.

You can create text in a file line by line, just as you would on a typewriter. However, **ed** is easier to use than a typewriter because it gives you commands that allow you to change, delete, or add text on several lines in the file, and then display those lines of text on your terminal. You can also add text from another file.

After you have read through this tutorial and have done the examples and exercises, you will have a good working knowledge of **ed**. The following basics will be covered:

- A brief introduction to **ed**, accessing the line editor, creating some text, displaying the lines of text, deleting lines, writing the text to a UNIX system file, and quitting **ed**,

- How to address those lines of the file that you want to work on,

- How to display lines of text,
- How to create text,
- How to delete text,
- How to substitute new text for old text,
- How to use special characters as shortcuts for search and substitute patterns,
- How to move text around in the file, and
- Some other useful commands and information.

HOW TO READ THIS TUTORIAL

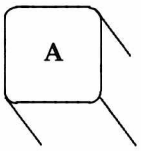
In this tutorial, commands printed in **bold** should be typed into the system exactly as shown. The system responses to those commands are shown in *italic*. Text that you type into a file is not shown in bold. You should assume that each line you type in at your terminal ends in a carriage return unless the text directs you to do something else. The carriage return is denoted by **<CR>**. As you read the text, you may want to glance back to this section for a quick recap of these conventions.

bold command	(Type in exactly as shown.)
italic response	(The system's response to the command.)
roman	(Text that is being typed into a file.)
<CR>	(Carriage return.)

A display screen or partial screen, like the one above, will be used to illustrate the commands. Because **ed** is versatile and can be used on any type of terminal, you may not be working on a video display

terminal. However, the lines you type in, and the system responses are the same whether you are working with a video display terminal or a paper printing terminal.

The **ed** commands are introduced by depicting the corresponding key on your keyboard. The key will appear as shown below in the example of the "a" key.



Notice that the letter on the key appears as it does on your keyboard. However, when you press the key, the letter will appear in lowercase on your terminal. If you need an uppercase letter, the example will include the SHIFT key.

The commands discussed in each section are reviewed at the end of that section. A summary of the **ed** commands discussed in this chapter is found in *Appendix D*, where they are listed in alphabetical order, as well as by topic.

At the end of some sections, exercises are given so you can experiment with the commands. The answers to all of the exercises are at the end of this chapter.

GETTING STARTED

Let's get started. The best way to learn **ed** is to log into the UNIX system and try the examples as you read this tutorial, do the exercises, and do not be afraid to experiment with the **ed** commands. The more you experiment with **ed** commands, the sooner these commands will become second nature to you, and you will have a fast and versatile method of editing text.

In this section, you will learn the bare essentials on how to:

- Access **ed**,
- Append some text,
- Move up or down in the file to display a line of text,
- Delete a line of text,
- Write the buffer to a file, and
- Quit **ed**.

How to Access ed

To access the line editor, type in **ed** and then a file name. The general format for the **ed** command line is:

ed filename<CR>

Choose a file name that reflects what will be in the file. The system will respond with a question mark if this is a new file.

```
$ ed new-file<CR>
? new-file
```

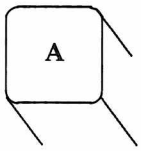
If you are going to edit an existing file, **ed** will respond with the number of characters in the file.

```
$ ed old-file<CR>
235
```

In the above example, the existing file, *old-file*, has 235 characters.

How to Create Text

If you have just accessed **ed**, you are in the command mode of the line editor. **ed** is waiting for your commands. How do you tell **ed** to create some text? Press the "a" key and then a carriage return.



Append text.

If **a** is the only character on a line, it tells the editor that the next characters typed in from the terminal are text for the file. You are now in the text input mode of **ed**. After you have added all the text that you want to the file, type in a period on the line by itself. This takes you out of the text input mode and returns you to the command mode of **ed**, so that you can give **ed** other commands.

The next example shows how to enter **ed** and begin creating text in the new file, *try-me*. The text input mode is then ended with a period.

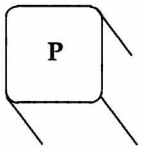
```
$ ed try-me<CR>
? try-me
a<CR>
This is the first line of text.<CR>
This is a second line,<CR>
and this is the third line.<CR>
.<CR>
```

Notice that **ed** does not give you a response to the period. It just waits for you to enter a new command. If **ed** is not responding to your commands, you may have forgotten to type in the period. Even experienced users sometimes forget to end the text input mode with a period. Type in a period at the beginning of the line. Now **ed** should respond to your commands. If you have added some

unwanted characters or lines to your text, you can delete them once you are back in the command mode.

How to Display a Line of Text

How can you display what is in the file? Type in **p**, for print, on a line by itself.



Display text.

Since you have not specified any line number, or line address, **p** will display the current line, that is, the line that was last touched or worked on by **ed**.

```
$ ed try-me<CR>
? try-me
a<CR>
This is the first line of text.<CR>
This is a second line,<CR>
and this is the third line.<CR>
.<CR>
p<CR>
and this is the third line.
```

If you want to see all the lines of text in the file, type in **1,$p**. The **1** and the **$** are the line addresses for the first line and the last line of the file. These will be discussed in detail in the section on *Line Addressing*.

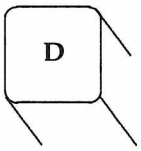
```
1,$p<CR>
This is the first line of text.
This is a second line,
and this is the third line.
```

Problem:
If you forgot to end the text input mode with the period, you would have added a line of text that you did not want. Try to make this mistake. Add another line of text to your *try-me* file and then try the **p** command without ending the text input mode. Now, end the text input mode and press "p". What did you get? How do you get rid of that line?

```
p<CR>
and this is the third line.
a<CR>
This is the fourth line.<CR>
p<CR>
.<CR>
1,$p<CR>
This is the first line of text.
This is a second line,
and this is the third line.
This is the fourth line.
p
```

How to Delete a Line of Text

If you are in the command mode of **ed**, press **d** to delete the current line.



Delete text.

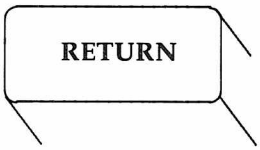
To get rid of the line with the "p" on it, in the last example, delete the line with the **d** command. The next example displays the current line, deletes the current line, and then displays all the lines in the file.

```
p<CR>
p
d<CR>
1,$p<CR>
This is the first line of text.
This is a second line,
and this is the third line.
This is the fourth line.
```

After you press **d**, **ed** deletes the current line, but it does so quickly and quietly. It is not evident to you that anything has happened unless you press **p** and find that the current line has been deleted.

How to Move Up or Down a Line in the File

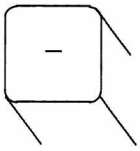
To display the line below the current line, press <CR>



Display the next line of text.

If there is no line below the current line, **ed** will respond with a ? and the current line will remain the last line of file. Pressing <CR> is a good way to move down through the buffer.

How do you display the line above the current line? Use the minus key, – .



Display the line of text above the current line.

The next screen demonstrates how to display a line of text, above or below the current line in the file.

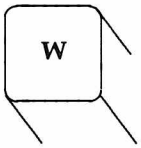
```
p<CR>
This is the fourth line.
–<CR>
and this is the third line.
–<CR>
This is a second line,
–<CR>
This is the first line of text.
<CR>
This is a second line,
<CR>
and this is the third line.
```

If you pressed the **−<CR>** or **<CR>**, you noticed that the line was displayed without having to press the "p" key. You were addressing a line. If you give a line address and do not follow it with a command, **ed** assumes you want the **p** command, which is the default command for a line address.

Experiment with these commands, create some text, delete a line, and display your file.

How to Save the Buffer Contents in a File

If you have finished editing your text, how do you move it from the buffer, your scratch pad, into a file? To save your text, write the contents of the buffer into a file with the **w** command.



Write the contents of the buffer to a file.

ed will remember the file name you gave when you accessed **ed**, and will write the contents of the buffer to a file with that name. If the file did not already exist, **ed** will create it and then write the contents of the buffer into it.

w<CR>
107

If the write command is successful, the character count is displayed. In the last example, there are 107 characters of text. When you write a file, you copy the contents of the buffer into the file. The text in the buffer is not disturbed. You can add more text to it. It is a good idea to write the buffer text into your file frequently. If an interrupt occurs (such as an accidental loss of power to your terminal), you may lose the material in the buffer, but you will not lose the copy written to your file. You can also write to another file name that is different

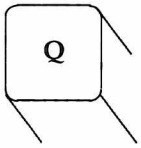
from the one you entered in the **ed** command line. The file name will be a parameter to the **w** command. In the following example, the new file name is *stuff*.

```
w stuff <CR>
107
```

When you return to the shell command mode, display the contents of *stuff* and *try-me*. Are they the same file?

How to Quit the Editor

You have completed editing your file, and have written the editing buffer to the file. To leave the editor and return to the shell command mode, type in the quit command, **q**.



Quit the editing buffer.

```
w<CR>
107
q<CR>
$
```

The system responds with your shell prompt. At this point, the editing buffer vanishes. Unless you have used the write command,

your text in the buffer has also vanished. Since this could be a serious problem, **ed** warns you with a ? the first time you type in **q** without having written any new changes to a file.

```
q<CR>
?
w<CR>
107
q<CR>
$
```

If you insist on typing in a second **q**, **ed** assumes you do not want to write the changes to the buffer into your file, and returns you to the shell command mode. Your file is left unchanged and the buffer contents are wiped out.

You now know the basic commands to create and edit a file.

SUMMARY OF COMMANDS FOR GETTING STARTED	
ed filename	Enter **ed** to edit the file called *filename*.
a	Append text after the current line.
.	End the text input mode, and return to the command mode of **ed**.
p	Display text on your terminal.
d	Delete text.
<CR>	Display the next line in the buffer.
–	Display the line above the current line in the buffer.
w	Write the buffer to the file.
q	Quit **ed** and return to shell command mode.

EXERCISE 1

The answers to all the exercises throughout this chapter are found at the end of this chapter. However, if your method works, if it performs the task even though it does not match the answer given, it is a correct answer.

1-1. Enter **ed** with the file named *junk*. Create a line of text "Hello World", write to the file and quit **ed**.

1-2. Reenter **ed** with the file named *junk*. What was the system response? Was it the same character count as the response to the **w** command in Exercise 1-1.?

Display the contents of the file. Is that your file *junk*?

How do you get back to the shell command mode? Try **q** without writing the file. Why do you think the editor allowed you to quit without writing to the buffer?

1-3. Enter **ed** with the file *junk*. Add a line:

This is not Mr. Ed, there is no horsing around.

Since you did not specify a line address, where do you think the line was added to the buffer? Display the contents of the buffer. Try quitting the buffer without writing to the file. Try writing the buffer to a different file *stuff*. Notice that **ed** does not warn you that the file *stuff* already exists. You have erased the contents of *stuff* and replaced it with new text.

GENERAL FORMAT OF ed COMMANDS

The commands in **ed** have a simple and regular format. Commands are of the form:

[address1,address2]command[parameter]<CR>

The brackets around the addresses and parameter denote that these are optional. The brackets are not part of the command line.

address1,address2
The addresses give the position of lines in the buffer. Address1 through address2 gives you a range of lines that will be affected by the command.

command
The command is one character and tells the editor what task to perform.

parameter
The parameters to a command are those parts of the text that will be modified, or a file name, or another line address.

This general format will become clearer to you when you begin to experiment with the commands in **ed**.

LINE ADDRESSING

Line addresses are very important to **ed**. To add text before or after a line, to delete, move, or change a line, **ed** must know the line address.

[address1,address2]command<CR>

Address2 is given only if you are specifying a range of lines. If address1 is not given, **ed** assumes that the line address is the current line.

A line address is a character or group of characters that identify a line of text. The most common ways to address a line in **ed** are:

- Line numbers, 1 being the first line of the file,
- Special symbols for the current line, last line, and a range of lines,

- Adding or subtracting a number of lines from the current line, and
- A character string or word on that line.

You can access one line, a range of lines, or make a global search for all lines containing a specified character string. A character string is a group of successive characters, such as a word.

Number Line Addresses

ed gives a number address to each line in the buffer. The first line of the buffer is 1, the second line of the buffer is 2 and so on for each line in the buffer. Each line can be accessed by **ed** with the line address number. If you want to see how line numbers address a line, enter **ed** with the file *try-me* and type in a number of a line.

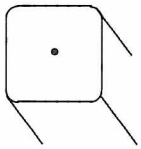
```
$ ed try-me<CR>
107
1<CR>
This is the first line of text.
3<CR>
and this is the third line.
```

Remember that **p** is the default command for **ed**. Since you gave a line address, **ed** assumes you wanted that line displayed on your terminal.

Problem:
Later in this tutorial you will create lines in the middle of the text, or delete lines, or move a line to a different position. This will change the address number of a line. The number of a specific line is always the current position of that line in the editing buffer. If you add five lines of text between line 5 and line 6, once the lines have been added, line 6 becomes line 11. If you delete line 5, line 6 becomes line 5.

Special Symbols Addresses

Current Line Address Character



The address of the current line.

The current line is the line that was most recently acted upon by **ed**, either displayed, created, or moved. If you have just accessed **ed** with an existing file, the current line is the last line of the buffer. The address for the current line is a period. If you want to display the current line, type in: **.**

If you access **ed** with your file *try-me*, you will find that the current line is the last line. Try it.

```
$ ed try-me<CR>
107
.<CR>
This is the fourth line.
```

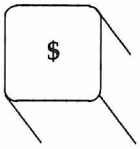
The "." is the address. Since no command is given, **ed** assumes the default command **p** and displays the line addressed by " **.** ".

If you want to know the line number of the current line, you can type in the command: **.=**

ed will respond with the line number. In the last example the current line is 4.

```
.<CR>
This is the fourth line.
.=<CR>
4
```

Last Line Address Character



The address of the last line.

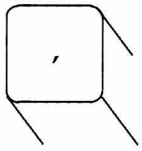
The last line of the file can be addressed by **$**. It does not matter how many lines are in the file, the last line can always be addressed by **$**. If you access **ed** with the *try-me* file, you can see that when you first enter **ed** the current line is the last line.

```
$ ed try-me<CR>
107
.<CR>
This is the fourth line.
$<CR>
This is the fourth line.
```

Remember that the **$** address within **ed** is not the same as the *$* prompt of the shell. If this gets confusing and you want to change your prompt, see *Changing Your Environment* in *Chapter 7, Shell Tutorial.*

Address for the First Line Through the Last Line

The , used as an address will refer to all lines of the file, the first line through the last line.



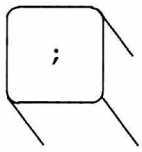
Address all lines of the file.

If you wanted to display all lines of the file, you could use , as a shortcut address for **1,$**.

```
,p<CR>
This is the first line of text.
This is a second line,
and this is the third line.
This is the fourth line.
```

Address for the Current Line Through the Last Line

The ; addresses the current line through the last line of the file.



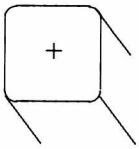
Address the range of lines from the current line through the last line.

The ; is the same as addressing **.,$**.

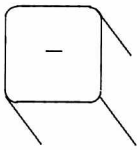
.<CR>
This is a second line,
;p<CR>
This is a second line,
and this is the third line.
This is the fourth line.

Relative Addressing, Adding or Subtracting Lines from the Current Line

If you are in a long file, you may want to address lines with respect to the current line. You can do this by adding or subtracting the number of lines from the current line, thus giving a relative line address.



Add a number of lines to the current line address.



Subtract a number of lines from the current line address.

To see relative line addressing, add several more lines to your file *try-me*. Each line should contain the number of the line.

```
$ ed try-me<CR>
107
.<CR>
This is the fourth line.
a<CR>
five
six
seven
eight
nine
ten
.<CR>
```

Now try adding and subtracting line numbers from the current line.

```
4<CR>
This is the fourth line.
+3<CR>
seven
-5<CR>
This is a second line,
```

What happens if you ask for a line address that is greater than the last line, or you try to subtract a number greater than the current line number? Experiment with a relative line addressing. See what happens.

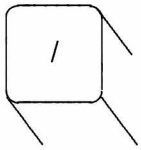
```
5<CR>
five
-6<CR>
?
.=<CR>
5
+7<CR>
?
```

Notice in the above example that the current line remains at line 5 of the buffer. The current line only changes if you give **ed** a correct address. The *?* response indicates an error. The section on *Other Useful Commands and Information* at the end of this chapter, will discuss getting a help message which describes the error.

Character String Addresses

You can search forward or backward in the file for a line containing a specified character string. The line address is the search delimiter and the character string.

A delimiter gives the boundaries of the character string. Delimiters tell **ed** where a character string starts and ends. The most common delimiter is /. You may also use **?**. If / is used at the beginning of an address **ed** will search forward or down the buffer for the next line containing the specified character string.

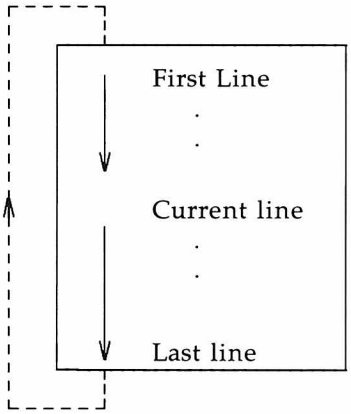


Search down or forward in the buffer and address the first line with a specified pattern of characters.

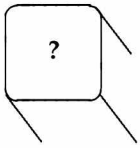
Type in: **/pattern**

ed will search the current line and then down the buffer for the first line that contains the characters **pattern**. If the search reaches the last line of the buffer, **ed** will then wrap around and start searching down the buffer from line 1.

The rectangle below represents the editing buffer. The path of the arrows shows the search initiated by / .



If **?** is used at the beginning of an address, **ed** will search backward or up in the buffer for the specified character string.

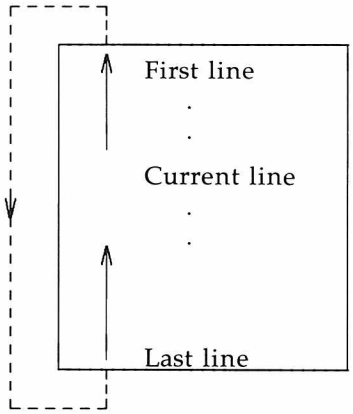


Search up or backward in the buffer and address the first line containing a specified pattern of characters.

Type in: **?pattern**

ed searches backward from the current line for the first line containing the characters **pattern**. If the search reaches the first line of the file, it will wrap around and continue searching upward from the last line of the file.

The next rectangle represents the editing buffer. The path of the arrows shows the search initiated by **?** .



Experiment with these two search address requests on the file *try-me*. What happens if **ed** does not find the search pattern?

```
$ ed try-me<CR>
107
.<CR>
ten
?first<CR>
This is the first line of text.
/fourth<CR>
This is the fourth line.
/junk<CR>
?
```

Once again, since no command was given, **ed** assumes it is the **p** command and displays the line. In the above example when **ed** was asked to search for the pattern **junk**, it could not find **junk** and responded with a ? .

Try the following sequence of commands.

Type in: **/line<CR>**
/<CR>

What happened?

.<CR>
This is the first line of text.
/line<CR>
This is the second line,
/<CR>
and this is the third line.
/<CR>
This is the fourth line.

ed remembers the pattern of the last search and looks for that pattern until it is given a new pattern.

Specifying a Range of Lines

There are two ways to address a range of lines. You can specify a range of lines such as address1 through address2, or you can specify a global search for all lines containing a specified pattern.

The simplest way to specify a range of lines is to use the line number of the first line through the line number of last line of the range. These numbers are separated by a comma and placed before the command. If you want to display lines four through ten of the editing buffer, you would give address1 as 4 and address2 as 10.

Type in: **4,10p<CR>**

If you are editing the file *try-me*, how would you display lines one through five?

```
1,5p<CR>
This is the first line of text.
This is a second line,
and this is the third line.
This is the fourth line.
five
```

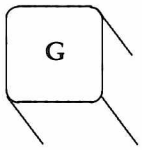
Did you try typing in **1,5** without the **p**? What happened? If you do not add the **p** command, **ed** only prints out address2, the last line of the range of addresses.

You can also use relative line addressing for a range of lines. Be careful, address1 must come before address2 in the buffer. The relative addresses are calculated from the current line.

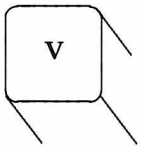
```
.<CR>
This is the fourth line
-2,+3p<CR>
This is a second line,
and this is the third line.
This is the fourth line.
five
six
seven
```

Specifying a Global Search

There are two commands that do not follow the general format of the **ed** commands. They are the global search commands that specify the addresses with a character string.



The global search command searches the entire file for lines that contain a specified pattern of characters.



The global search command searches the entire file for lines that do NOT contain a specified pattern of characters.

The general format for these two commands gives the command, a delimiter, the search pattern, a delimiter, and a command.

g/pattern/command<CR>
v/pattern/command<CR>

Try out these commands on *try-me*.

```
g/line/p<CR>
This is the first line of text.
This is a second line,
and this is the third line.
This is the fourth line
```

```
v/line/p<CR>
five
six
seven
eight
nine
ten
```

p will act as a default command for the lines addressed by **g** or **v**. If you just want to display the lines, you do not need the last delimiter or **p**.

```
g/line<CR>
This is the first line of text.
This is a second line,
and this is the third line.
This is the fourth line
```

If the lines are used as addresses for other **ed** commands, you will need the beginning and ending delimiters. All of these methods of addressing a line can be used as addresses for **ed** commands.

SUMMARY OF LINE ADDRESSING

1,2...	The number of the line in the buffer.
.	The current line, the last line **ed** touched.
.=	The command that gives the line number of the current line.
$	The last line of the file.
,	Addresses lines 1 through the last line.
;	Addresses the current line through the last line.
+ n	Add a number of lines *n* to the current line address.
− n	Subtract a number of lines *n* from the current line address.
/abc/	Search forward in the buffer and address the first line containing the pattern of characters *abc*.
?abc?	Search backward in the buffer and address the first line containing the pattern of characters *abc*.
g/abc/	Address all lines containing the pattern *abc*.
v/abc/	Address all lines that do NOT contain the pattern *abc*.

EXERCISE 2

2-1. Create a file *towns* with the following lines:

My kind of town is
Chicago
Like being no where at all in
Toledo
I lost those little town blues in
New York
I lost my heart in
San Francisco
I lost $$ in
Las Vegas

2-2. Display line 3.

2-3. What lines are displayed for the relative address range −2,+3p ?

2-4. The current line number is? Display the current line.

2-5. The last line says?

2-6. What line is displayed by the search:

?town<CR>

Now type in:

?<CR>

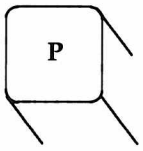
alone on a line. What happened?

2-7. Address all lines that contain the pattern "in". Then address all lines that do NOT contain the pattern "in".

DISPLAY LINES IN A FILE

The two commands that display lines of text in the editing buffer are **p** and **n**.

Display Lines of Text



Print or display lines of text in the editing buffer on your terminal.

You have already used the **p** command in several examples.

The general form of the print command is:

address1,address2p<CR>

p does not have parameters. However, it can be combined with the substitute command line. This will be discussed later in this chapter.

Experiment with different line addresses and the **p** command on a file in your directory. Try out the following types of addresses.

Type in: **1,$p<CR>**

The entire file should have been displayed on your terminal.

Type in: **−5p<CR>**

The editor should have subtracted 5 from the current line and displayed that line.

Type in: **+2p<CR>**

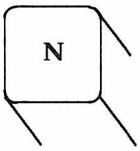
The editor should have added 2 to the current line and displayed that line.

Type in: **1,/a/p<CR>**

Did you figure out what happened? The editor searched for the next "a" from the current line, and then displayed lines 1 through the first line that contained "a" after the current line.

It is very important to delimit the search pattern to avoid errors in **ed**. You have to delimit the search pattern "a" (enclose "a" between slashes) so that **ed** can tell the difference between the search pattern address "a" and an **ed** command **a**.

Display Lines of Text Preceded by the Line Address Number



Display the line address number with the line of text.

The **n** command is a convenient command when you are deleting, creating, or changing lines. Besides displaying the lines of text, **n** precedes each line with the line address number.

The general format for **n** is the same as **p**.

[address1,address2]n<CR>

Also, like **p**, **n** does not have parameters, but it can be combined with the substitute command.

Try out **n** on your test file *try-me*.

```
$ ed try-me<CR>
137
1,$n<CR>
1       This is the first line of text.
2       This is a second line,
3       and this is the third line.
4       This is the fourth line.
5       five
6       six
7       seven
8       eight
9       nine
10      ten
```

Experiment with **n** using different line addresses. In the next example, the relative line addresses **−5** and **+2** are used. Also, the range of lines addressed from line 1 through the first line after the current line that contains an "ne" is also displayed.

```
−5n<CR>
5           five
+2n<CR>
7           seven
1,/ne/n<CR>
1           This is the first line of text.
2           This is a second line,
3           and this is the third line
4           This is the fourth line.
5           five
6           six
7           seven
8           eight
9           nine
```

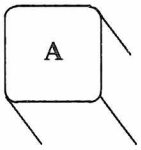
SUMMARY OF DISPLAY COMMANDS	
p	Displays on your terminal the specified lines of text in the editing buffer.
n	Displays on your terminal the line address numbers with the specified lines of text in the editing buffer.

CREATING TEXT

ed has three basic commands for creating new lines of text:

- **a** Append text,
- **i** Insert text, and
- **c** Change text.

Appending Text



Create text after the specified line in the buffer.

You have already used the append command in the *Getting Started* section of this tutorial. The general format for the append command is:

[address1]a<CR>

The default for address1 is the current line. If you do not give **a** an address, **ed** will make address1 the current line.

You have used the default address for **a**, now try using different line numbers for address1. In the next example, a new file called *new-file* is created. The first append command uses the default address. The second append command uses address1 as **1**. The lines are displayed with **n** so that you can see the line addresses.

```
$ ed new-file<CR>
?new-file
a<CR>
Create some lines
of text in
this file.
.<CR>
1,$n<CR>
1           Create some lines
2           of text in
3           this file.
1a<CR>
This will be line 2<CR>
This will be line 3<CR>
.<CR>
1,$n<CR>
1           Create some lines
2           This will be line 2
3           This will be line 3
4           of text in
5           this file.
```

Notice that the address of the line "of text in" changes from two to four after you append the two new lines.

Try out the following special addresses.

.a<CR>	Append after the current line.
$a<CR>	Append after the last line of the file.
0a<CR>	Append text before the first line of the file.

Each of these addresses is used to append text in the following examples.

```
.<CR>
This is the current line
.a<CR>
This line is after the current line.<CR>
.<CR>
-1,.p<CR>
This is the current line.
This line is after the current line.
```

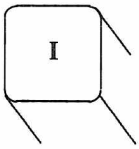
```
$a<CR>
This is the last line now.<CR>
.<CR>
$<CR>
This is the last line now.
```

```
0a<CR>
This is the first line now.<CR>
This is the second line now.<CR>
The line numbers change<CR>
as lines are added.<CR>
.<CR>
1,4n<CR>
1        This is the first line now.
2        This is the second line now.
3        The line numbers change
4        as lines are added.
```

The **0a** command can be replaced by the next command, the insert command.

Inserting Text

The insert command creates text before a specified line in the editing buffer.



Insert text before the specified line.

The general format for **i** is the same as for **a**.

[address1]i<CR>

As with the append command, you can insert one or more lines of text. The text input mode is always ended with a period alone on a line.

The example that follows inserts a line of text above line two; inserts a line of text above the first line; and displays all the lines of the buffer with **n**.

```
2i<CR>
Now this is line 2.<CR>
.<CR>
1,$n<CR>
1          Line 1
2          Now this is line 2
3          Line 2
4          Line 3
5          Line 4
1i<CR>
In the beginning<CR>
1,$n<CR>
1          In the beginning
2          Line 1
3          Now this is line 2
4          Line 2
5          Line 3
6          Line 4
```

Take a few minutes to experiment with the insert command. Try out the special line addresses.

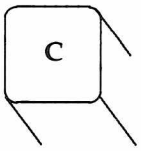
Type in: **.i<CR>**

or

Type in: **$i<CR>**

Changing Text

The change text command erases all of the specified lines and creates new text beginning at address1. You can create one or more lines of text. The change command puts you in the text input mode, so you must end the text input mode by a period alone on a line.



Erase specified lines and create new text.

Since **c** can erase a range of lines, the general format for the change command gives both address1 and address2.

[address1,address2]c<CR>

Address1 is the first line to be erased, and address2 is the last line of the range of lines to be replaced by new text. If you only want to erase one line of text, you would use only address1. If you do not type in address1, **ed** assumes the current line is address1.

The next example changes a range of lines. The first five lines are displayed with **n**. Then lines one through four **(1,4c)** are changed. The lines in the buffer are displayed after the change.

```
1,5n<CR>
1           Line 1
2           Line 2
3           Line 3
4           Line 4
5           Line 5
1,4c<CR>
Change line 1<CR>
and line 2 through 4<CR>
.<CR>
1,$n<CR>
1           Change line 1
2           and line 2 through 4
3           Line 5
```

Now experiment with **c**. Try changing the current line.

```
.<CR>
This is the current line.
c<CR>
I am changing the current line.<CR>
.<CR>
.<CR>
I am changing the current line.
```

If you are not sure you have left the text input mode, it is a good idea to type in the period a second time. If the current line is displayed, you know you are in the command mode of **ed**.

SUMMARY OF CREATE COMMANDS

---

a	Append text after the specified line in the buffer.
i	Insert text before the specified line in the buffer.
c	Change the text on the specified lines to new text
.	End the text input mode with a period alone on a line, and return to **ed** command mode.

EXERCISE 3

3-1. As an experiment, create a new file *ex3*. Instead of using the append command to create new text in the empty buffer, try the insert command. What happened?

3-2. Enter the file *towns* into **ed**. What is the current line?

Insert above the third line:

Illinois<CR>

Insert above the current line:

or<CR>
Naperville<CR>

Insert before the last line:

hotels in<CR>

Display the text in the buffer preceded by line numbers.

3-3. In the file *towns*, display lines one through five and replace lines two through five with:

London<CR>

Display lines one through three.

3-4. After you have completed exercise 3-3, what is the current line?

Find the line of text containing:

Toledo

Replace:
Toledo
with:
Peoria

Display the current line.

3-5. With one command line search for and replace:

New York
with:
Iron City

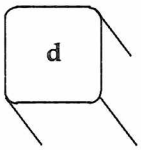
DELETING TEXT

This section of the tutorial discusses the delete commands:

d	Delete lines in the command mode;
u	Undo the last command;
# or <BACK SPACE>	Delete characters in the text input mode; and
@	Delete a line of text in the text input mode or delete the current command line.

Deleting Lines of Text

You have already deleted lines of text with the delete command **d** in the section of *Getting Started*.



Delete one or more lines of text.

The general format for **d** is:

[address1,address2]d<CR>

You can delete a range of lines, address1 through address2, or you can delete one line using only address1. If no address is given, **ed** assumes you want to delete the current line.

The next example displays lines one through five and then deletes the range of lines two through four.

```
1,5n<CR>
1          1 horse
2          2 chickens
3          3 ham tacos
4          4 cans of mustard
5          5 bails of hay
2,4d<CR>
1,$n<CR>
1          1 horse
2          5 bails of hay
```

How would you delete only the last line of a file?

```
$d<CR>
```

How would you delete the current line? One of the most common errors in **ed** is forgetting to end the create mode with a period. A line or two of text that you do not want may be added to the buffer. In the next example, the print command is accidentally added to the text before the create mode is ended. Then the current line, the print command, is deleted.

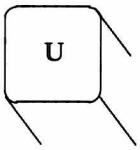
```
a<CR>
Last line of text<CR>
1,$p<CR>
.<CR>
p<CR>
1,$p
.d<CR>
p<CR>
Last line of text.
```

Remember that **1,$p** prints every line of the buffer.

Before you do much experimenting with the delete command, you may first want to learn about the **u** command.

Undo the Last Command

The undo command will erase the effect of the last command and restore any text that had been added, changed, or deleted by that command.



Undo the last command.

If you create new text, change lines of text, delete lines of text, or read new lines into the file, **u** undoes the effect of these commands. (The read command will be discussed in the section on *Moving Text*). Since **u** undoes the last command, it does not have any addresses or arguments. The general form is:

u<CR>

u does not undo the write command or the quit command. However, **u** will undo an undo command.

One example of the **u** command is restoring deleted lines. If you delete all the lines in the file and then type in **p, ed** will respond with a *?* since there are no more lines in the file. Type in **u** and all lines of the file will be restored.

```
1,$d<CR>
p<CR>
?
u<CR>
p<CR>
This is the last line
```

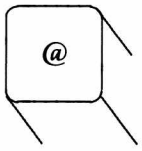
Now try **u** on the append command.

```
.<CR>
This is the only line of text
a<CR>
Add this line<CR>
.<CR>
1,$p<CR>
This is the only line of text
Add this line
u<CR>
1,$p<CR>
This is the only line of text
```

Deleting Commands in the Text Input Mode

Deleting the Current Line

The @ will delete the current line of typing. The line will not be erased from your terminal, but will end with an @ sign and the cursor will move to the next line. When you end the create mode and display the lines of text, the deleted line will not appear.



Delete the current line in the text input mode.

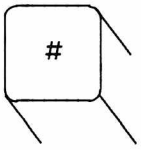
```
a<CR>
I don't want to add this @
a new line of text<CR>
.<CR>
1,$p<CR>
a new line of text
```

The above example begins creating a new file. The first line is deleted in the text input mode, therefore, only the second line is displayed by the **1,$p** command. **@** will also delete the current command line. If you make an error typing in a command, type in **@** instead of **<CR>** and **ed** will ignore the command. In the next example, an incorrect address is given, so the command line is cancelled with **@**.

```
1,$d@
1d<CR>
```

Deleting the Last Characters Typed

If you only made a mistake in typing the last few characters, the **#** or **<BACK SPACE>** can delete those characters if you have not pressed **<CR>**.



Delete the last character just typed into the buffer.

Delete the last character just typed into the buffer.

The **<BACK SPACE>** key will delete characters if you have changed your environment to include this command. (See *Chapter 7, Shell Tutorial* for changing your environment.)

```
a<CR>
This is a typoo#<CR>
.<CR>
.<CR>
This is a typo
```

In the above example, the extra o in typo was deleted by #. When the line is displayed the error is gone.

You must enter a # for each character that needs to be erased or retyped. In the following example, the error is corrected and new characters follow the last #. (The **<BACK SPACE>** will back up over the characters.)

```
a<CR>
To the IRS, I mail a check<CR>
for one hun###thousand dollars.<CR>
.<CR>
.<CR>
for one thousand dollars.
```

If you press **<CR>** before you correct the error, it is too late to correct the error in the text input mode. However, once you have left the text input mode, the substitute command, discussed in the next section, can solve your problem.

Create a junk file and practice each of these four commands until you are comfortable with them.

SUMMARY OF DELETE COMMANDS

In the command mode:

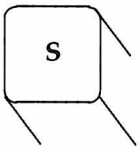
d	Delete one or more lines of text.
u	Undo the last command.
@	Delete the current command line.

In the text input mode:

@	Delete the current line.
# or **<BACK SPACE>**	Delete the last character typed in.

SUBSTITUTING TEXT

You can modify your text with the substitute command **s**.



Replace a pattern of characters with new text.

The substitute command replaces the first occurrence of a string of characters with new text. The general format is:

[address1,address2]s/old text/new text/[command]<CR>

Since this is a more complicated format than the preceding commands, let's look at it piece by piece.

address 1 and address2

The range of lines being addressed by **s**. The address can be one line, address1, a range of lines address1 through address2 or the global search address. If no address is given, **ed** will make the substitution on the current line.

s

The substitute command, which is positioned right after the line address.

/old text/

The text to be replaced. It is usually delimited by backslashes, however, it can be delimited by other symbols such as **?** or a period. The **old text** matches the first occurrence of the words or characters to be replaced.

/new text/

The text that replaces the **old text**. It is placed between the second and third delimiters and replaces the **old text** between the first and second delimiters.

command

This may be one of four commands that can be placed after the last delimiter. The commands are:

g Change all occurrences of **old text** on the specified lines.

l Display the last line of substituted text including nonprinting characters. (See last section of this chapter entitled *Other Useful Commands and Information.*)

n Display the last line of the substituted text preceded by the line number.

p Display the last line of substituted text.

Substituting on the Current Line

The simplest example of the substitute command is making a change to the current line. You do not need to give the line address for the current line.

s/old text/new text/<CR>

In the next example, a typing error was made on the current line. The example displays the current line, then makes the substitution to correct the error. The **old text** is the **ai** of **airor**, the **new text** is **er**.

```
.p<CR>
In the beginning, I made an airor
s/ai/er/<CR>
.p<CR>
In the beginning, I made an error
```

Did you try out the example? Did you notice **ed** was quiet and gave no response to the substitute command? You either have to display the line with **p** or **n**, or place **p** or **n** on the substitute line. The example below substitutes file for toad.

```
.p<CR>
This is a test toad
s/toad/file/n<CR>
1          This is a test file
```

ed has a short cut for you. If you leave off the last delimiter of the substitute command, the line will automatically be displayed.

```
.p<CR>
This is a test file
s/file/frog<CR>
This is a test frog
```

Substituting on One Line

To substitute on a line that is not the current line, use address1.

[address1]s/old text/new text/<CR>

In this example, the current line is line three. Line one will be corrected.

```
1,3p<CR>
This is a pest toad
testing testing
come in toad
.<CR>
come in toad
1s/pest/test<CR>
This is a test toad
```

Notice that the last delimiter was omitted and **ed** printed out the line.

Substituting on a Range of Lines

If you want to make a substitution on a range of lines, you can specify the first address, address1, through the last address, address2.

[address1,address2]s/old text/new text/<CR>

If **ed** does not find the pattern to be replaced on one of the lines, no changes are made to that line. In the next example, all the lines in

the file are addressed for the substitute command. However, only the lines that contain the old text, **es**, are changed.

```
1,$p<CR>
This is a test toad
testing testing
come in toad
testing 1, 2, 3
1,$s/es/ES/n<CR>

4          tESting 1, 2, 3
```

When you specify a range of lines, **p** or **n** on the substitute line only prints out the last line changed.

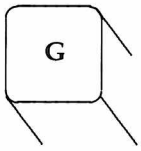
To display all the text that was changed use **n** or **p** alone in a command line.

```
1,$n<CR>
1          This is a tESt toad
2          tESting testing
3          come in toad
4          tESting 1, 2, 3
```

Notice only the first occurrence of "es" is changed on line 2. How do you change every occurrence?

Global Substitution

One of the most versatile tools in **ed** is global substitution.



Global substitution or search.

If you place the **g** command after the last delimiter of the substitute command, you will change every occurrence on the specified lines. Try changing every occurrence of **es** in the last example. If you are following along, doing the examples as you read this, remember you can use **u** to undo the last substitute command.

```
u<CR>
1,$p<CR>
This is a test toad
testing, testing
come in toad
testing 1, 2, 3
1,$s/es/ES/g<CR>
1,$p<CR>
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

Another way to do the above example is to use the global search as an address instead of the range of lines one through the last line (**1,$**).

```
1,$p<CR>
This is a test toad
testing testing
come in toad
testing 1, 2, 3
g/test/s/es/ES/g<CR>
1,$p<CR>
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

If the global search pattern is unique, and is the same as the **old text** to be replaced, you can use an **ed** shortcut. You do not need to repeat the pattern for the **old text**. **ed** remembers the search pattern and uses it again as the pattern to be replaced.

g/old text/s//new text/g<CR>

```
1,$p<CR>
This is a test toad
testing testing
come in toad
testing 1, 2, 3
g/es/s//ES/g<CR>
1,$p<CR>
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

Experiment with the other search pattern addresses:

/pattern<CR>
?pattern<CR>
v/pattern<CR>

See how they react with the substitute command. In the example below, the **v/pattern** is used to locate the characters **in** that are NOT in the word **testing**.

```
v/testing/s/in/out<CR>
This is a test toad
come out toad
```

If you leave off the last delimiter all search addresses will print out including the ones where no substitution occurs.

```
g/testing/s//jumping<CR>
jumping testing
jumping 1, 2, 3
```

Notice that the global search substitutes for only the first occurrence of testing in each line. The lines are displayed on your terminal because the last delimiter is missing.

EXERCISE 4

4-1. In your file *towns* change **town** to **city** on all lines but the line with **little town** on it.

The file should read:

My kind of city is

London
Like being no where at all in
Peoria
I lost those little town blues in
Iron City
I lost my heart in
San Francisco
I lost $$ in
hotels in
Las Vegas

4-2. Try using **?** as a delimiter. Change the current line

Las Vegas

to

Toledo

You could also use the change command **c**, since you were changing the whole line.

4-3. Try searching backward in the file for the word

lost

and substitute

found

using the **?** as the delimiter. Did it work? (The last line of the file is the current line.)

4-4. Search forward in the file for

no

and substitute

NO

for it. What happens if you try to use **?** as a delimiter?

Experiment with the various combinations of addressing a range of lines and global searches.

What happens if you try to substitute for the **$$** ? Try to substitute for the **$** on line nine of your file.

Type in: **9s/$/Big $<CR>**

What happened?

9s/$/Big$<CR>
I found $$ in Big $

The substitution did not work correctly because **$** is a special character in **ed**. It will be discussed next in the section on special characters.

SPECIAL CHARACTERS

If you tried to substitute for the **$** in the line

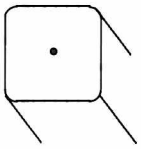
I lost my $ in Las Vegas

you would find that instead of replacing the **$**, the new text was placed at the end of the line. The **$** is a special character meaning the end of the line.

ed has several special characters that give you a shorthand for search patterns and substitution patterns. The characters act as wild cards. If you have tried to type in any of these characters, the result was probably different than what you had expected.

The special characters are:

.	Match any one character.
*	Match zero or more occurrences of the preceding character.
.*	Match zero or more occurrences of any character following the period.
^	Match the beginning of the line.
$	Match the end of the line.
\	Take away the special meaning of the special character that follows.
&	Repeat the old text to be replaced in the new text of the replacement pattern.
[...]	Match the first occurrence of a character in the brackets.
[^...]	Match the first occurrence of a character that is NOT in the brackets.



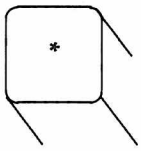
Match any one character.

The period will represent any one character in a search or substitute pattern. In the next example, a list of animals is searched for the pattern of any letter followed by **at**.

```
1,$p<CR>
rat
cat
turtle
cow
goat
g/.at<CR>
rat
cat
goat
```

Notice that the characters **oat** in goat match **.at**.

The combination of the period and the * is a very potent wild card for the substitution pattern. (See below)



Match zero or more occurrences of the preceding character.

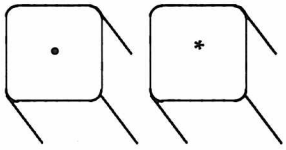
The * is shorthand for a character that is repeated several times in a row in a search or substitute pattern. For example, if you were creating some text and held down a key a little too long, the character would be entered several times into your text. The * is an easy way to substitute one character for those extra characters.

```
p<CR>
brrroke
s/br*/br<CR>
broke
```

It is important to include the **b** in the substitute pattern since * will substitute for *zero* or more occurrences of **r**. Below is an example of using only **r***.

p<CR>
brrroke
s/r*/r<CR>
rbrrroke

The first zero or more occurrences of r is at the beginning of the line where there are no occurrences of r.



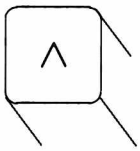
Match zero or more occurrences of any character after the period.

If you combine the period and the *, the combination will match all characters after the period. With this combination you can replace all characters on the last part of a line.

p<CR>
Toads are slimy, cold creatures
s/are.*/are wonderful and warm<CR>
Toads are wonderful and warm

The .* can also replace all characters between two patterns.

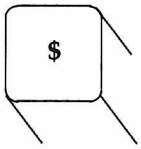
p<CR>
Toads are slimy, cold creatures
s/are.*cre/are wonderful and warm cre<CR>
Toads are wonderful and warm creatures



Match the beginning of a line.

If you want to insert a word at the beginning of a line, use the ^ for the old text to be substituted. This is very helpful when you want to insert the same pattern in the front of several lines. The next example places the word **all** at the beginning of each line.

```
1,$p<CR>
creatures great and small
things wise and wonderful
things bright and beautiful
1,$s/^/all /<CR>
1,$p<CR>
all creatures great and small
all things wise and wonderful
all things bright and beautiful
```



Match the end of the line.

This character is useful for adding characters at the end of a line or a range of lines.

```
1,$p<CR>
I love
I need
I use
The IRS wants my
1,$s/$/ money.<CR>
1,$p<CR>
I love money.
I need money.
I use money.
The IRS wants my money.
```

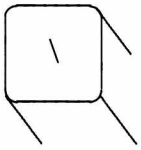
Did you try out the last two examples? Did you remember to put a space after the **all** or before **many**? **ed** adds the characters to the very beginning or the very end of the sentence. If you forgot the space before **money**, your file looks like the following:

```
1,$s/$/money/<CR>
1,$p<CR>
I lovemoney
I needmoney
I usemoney
The IRS wants mymoney
```

The **$** is a good way to add punctuation to the end of the line.

```
1,$p<CR>
I love money
I need money
I use money
The IRS wants my money
1,$s/$/./<CR>
1,$pf1/<CR>
I love money.
I need money.
I use money.
The IRS wants my money.
```

Since **.** is not matching a character, but replacing a character, it does not have a special meaning in this case. How could you change a period in the middle of a line to another punctuation? You must take away the special meaning of the period in the old text.



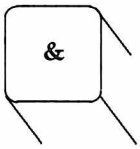
Take away the special meaning of the following special character.

If you want to substitute or search for some of the special characters, you must precede them by a \ . To change a **period**, precede the **.** with a \ .

```
p<CR>
Way to go.  Wow!
s/ \. /!<CR>
Way to go!  Wow!
```

Because the backslash is a special character, it too must be preceded by a \ if it is used in the old text.

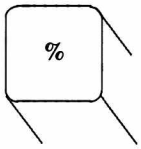
```
p<CR>
Way to go\  Wow!
s/ \\ /!<CR>
Way to go!  Wow!
```



Repeat the old text to be replaced in the new text of the replacement pattern.

If you want to add text without changing the rest of the line, the **&** is a useful shortcut. The **&** repeats the old text in the replacement pattern, so you do not have to worry about typing the correct pattern twice. The next screen shows an example of this.

```
p<CR>
The neanderthal skeletal remains
s/thal/& man's/<CR>
The neanderthal man's skeletal remains
```

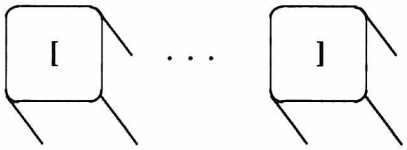


Repeat the last replacement pattern.

ed automatically remembers the last pattern of characters in a search pattern or the old text in a substitution. But, you must tell **ed** to repeat the replacement characters in a substitution with the **%**. The **%** pattern is very useful if you do not want to make a global change, but you do want to make the same substitution on several different lines. If you want to change money into gold for yourself, but not the IRS, you would repeat the last substitution from line one on line three, but not on line four.

```
1,$p<CR>
I love money
I need food
I use money
The IRS wants my money
1s/money/gold<CR>
I love gold
3s//%<CR>
I use gold
1,$p<CR>
I love gold
I need food
I use gold
The IRS wants my money
```

ed automatically remembers **money**, the old text to be replaced, so it does not have to be repeated between the first two delimiters. The **%** tells **ed** to use the last replacement pattern, gold.



Match the first occurrence of a character in the bracket.

ed will try to match one of the characters enclosed in the brackets and substitute the specified old text with new text. The brackets can occur anywhere in the pattern to be replaced.

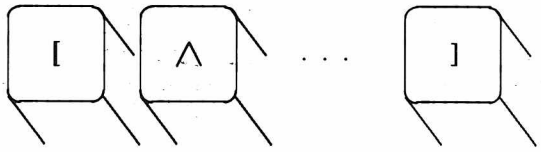
To conceal the large appetite of the anteater, the zoo keeper quietly altered his file on the animal's dietary habits as shown in the following screen.

```
1,$p<CR>
Monday        33,000 ants
Tuesday       75,000 ants
Wednesday     88,000 ants
Thursday      62,000 ants
1,$s/[6789]/4<CR>
Monday        33,000 ants
Tuesday       45,000 ants
Wednesday     48,000 ants
Thursday      42,000 ants
```

In the example above, the first occurrence of 6, 7, 8, or 9 was changed to 4 on each line that **ed** found a match.

The next example deletes the Mr or Ms from a list of names.

```
1,$p<CR>
Mr Arthur Middleton
Mr Matt Lewis
Ms Anna Kelley
Ms M. L. Hodel
1,$s/M[rs] //<CR>
1,$p<CR>
Arthur Middleton
Matt Lewis
Anna Kelley
M. L. Hodel
```



Match the first occurrence of a character that is not in the brackets.

If the caret is placed as first character in the brackets it tells **ed** to replace characters that are NOT one of these characters. However, if the caret is placed at any other position other than the first character, it will stand for the character ^.

If a copy of John's grades were sent to him as a file in his login, he could enter the file into **ed** and make the following changes to correspond with his own evaluation of his performance.

```
1,$p<CR>
grade  A  Computer Science
grade  B  Robot Design
grade  A  Boolean Algebra
grade  D  Jogging
grade  C  Tennis
1,$s/grade [^AB]/grade A<CR>
1,$p<CR>
grade  A  Computer Science
grade  B  Robot Design
grade  A  Boolean Algebra
grade  A  Jogging
grade  A  Tennis
```

Whenever you use special characters as wild cards in the old text to be changed, remember to use a unique pattern of characters. In the above example, if you had used only

1,$s/[^AB]/A<CR>

you would have changed the **g** in **grade** to A. Try it.

As with all commands in **ed**, experiment with these special characters. Find out what happens (or does not happen) if you use them in different combinations.

SUMMARY OF SPECIAL CHARACTERS

.	Match any one character in a search or substitute pattern.
*	Match zero or more occurrences of the preceding character in a search or substitute pattern.
.*	Match zero or more occurrences of any characters following the period.
^	Match the beginning of the line in the substitute pattern to be replaced or in a search pattern.
$	Match the end of the line in the substitute pattern to be replaced.
\	Take away the special meaning of the special character that follows in the substitute or search pattern.
&	Repeat the old text to be replaced in the new text replacement pattern.
%	Repeat the last replacement pattern.
[...]	Match the first occurrence of a character in the brackets.
[^...]	Match the first occurrence of a character that is NOT in the brackets.

EXERCISE 5

5-1. Create a file that contains the following lines of text.

```
A   Computer Science
D   Jogging
C   Tennis
```

What happens if you try the command line:

1,$s/[^AB]/A/<CR>

Undo the above command. How would you make the C and D unique? (Hint: they are at the beginning of the line ^.) Do not be afraid to experiment!

5-2. Insert the following line above line 2:

These are not really my grades

Using brackets and the beginning of the line character ^, create a search pattern that you could use to locate the line you inserted. There are several ways to address a line. When you edit text, use the way that is quickest and easiest for you.

5-3. With one command, change the next three lines

I love money
I need money
The IRS wants my money

to the following lines:

It's my money
It's my money
The IRS wants my money

Using two command lines: change the first line from money to gold, change the last two lines from money to gold without using the characters **money** or **gold**.

5-4. How would you change the line

1020231020

to

10202031020

without repeating the old digits in the replacement pattern?

5-5. Create a line of characters

```
* . \ & % ^ *
```

Substitute a letter for each character. Did you need to use the backslash for every substitution?

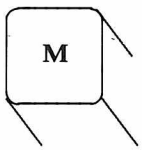
MOVING TEXT

You have now learned to address lines, create and delete text, and make substitutions. **ed** has one more set of versatile and important commands. You can move, copy, or join lines of text in the editing buffer. You can also read in text from a file that is not in the editing buffer, or write lines of the file in the buffer to another file in the current directory. The commands that move text are:

m	Move lines of text.
t	Copy lines of text.
j	Join contiguous lines of text.
w	Write lines of text to a file.
r	Read in the contents of a file.

Move Lines of Text

You can move paragraphs of text to another place in the file, or you can move an entire subroutine of a program to another place in the computer program you are creating in **ed**.



Move one or more lines of text.

The general format for the move command is:

[address1,address2]m[address3]<CR>

address1,address2
The range of lines to be moved. If only one line is moved, only address1 is given. If no address is given, the current line is moved.

m The move command.

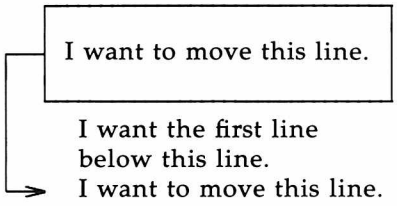
address3 Place the text after this line.

The following lines are in a file.

I want to move this line.
I want the first line
below this line.

Type in: **1m3<CR>**

ed will move line 1 below line 3.

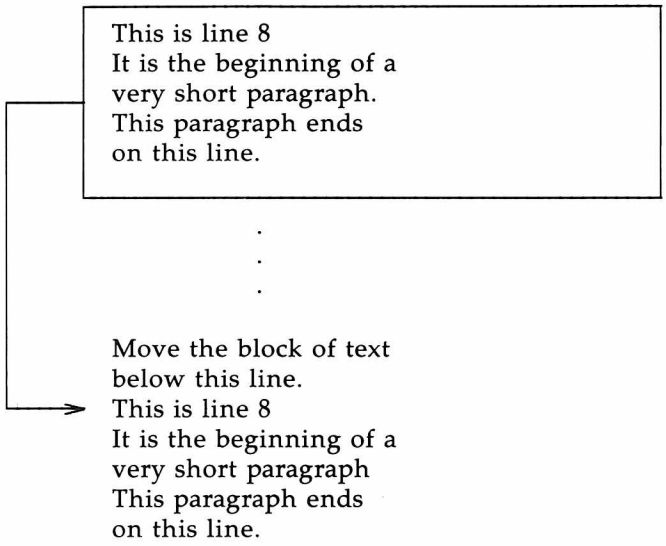


The next screen shows how this will appear on your terminal.

1,$p<CR>
I want to move this line.
I want the first line
below this line.
1m3<CR>
1,$p<CR>
I want the first line
below this line.
I want to move this line.

If you want to move a paragraph of text, address1 and address2 would be the range of lines of the paragraph.

The following example depicts moving a block of text. Line 8 through line 12 are moved below line 65.



The next screen shows how the command would appear on your terminal. The **n** command is used so that you can see how the line numbers change.

```
8,12n<CR>
8          This is line 8.
9          It is the beginning of a
10         very short paragraph.
11         This paragraph ends
12         on this line.
64,65n<CR>
64         Move the block of text
65         below this line.
8,12m65<CR>
59,65n<CR>
59         Move the block of text
60         below this line.
61         This is line 8.
62         It is the beginning of a
63         very short paragraph.
64         This paragraph ends
65         on this line.
```

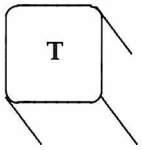
How do you think you would move lines above the first line of the file? Try the following command.

Type in: **3,4m0<CR>**

When address3 is 0, the lines are placed at the beginning of the file.

Copy Lines of Text

The copy command **t** acts like the **m** command except that the block of text is not deleted at the original address of the line. A copy of that block of text is placed after a specified line of text.



Copy lines of text and place them after a specified line.

The general format of the **t** command also looks like the **m** command.

[address1,address2]t[address3]<CR>

address1,address2
The range of lines to be copied. If only one line is copied, only address1 is given. If no address is given, the current line is copied.

t The copy command.

address3 Place the copy of the text after this line.

You may want to reiterate a set of directions. You can place a copy of those lines of text below another line in the file. In the next example you want to copy three lines of text below the last line.

```
                    Safety procedures:

  If there is a fire in the building:
  Close the door of the room to seal off the fire

  +-------------------------------------+
  | Break glass of nearest alarm        |
  | Pull lever                          |
  | Locate and use fire extinguisher    |
  +-------------------------------------+
                .
                .
                .
  A chemical fire in the lab requires that you:

  +-------------------------------------+
  | Break glass of nearest alarm        |
  | Pull lever                          |
  | Locate and use fire extinguisher    |
  +-------------------------------------+
```

The commands and **ed**'s responses to those commands are displayed in the next screen. The **n** command displays the line numbers.

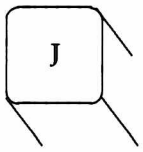
```
5,8n<CR>
5        Close the door of the room, to seal off the fire.
6        Break glass of nearest alarm
7        Pull lever
8        Locate and use fire extinguisher
30n<CR>
30       A chemical fire in the lab requires that you:
6,8t30<CR>
30,$n<CR>
30       A chemical fire in the lab requires that you:
31       Break glass of nearest alarm
32       Pull lever
33       Locate and use fire extinguisher
6,8n<CR>
6        Break glass of nearest alarm
7        Pull lever
8        Locate and use fire extinguisher
```

The text in lines six through eight remain in place. A copy of those three lines is placed after line 50.

Experiment with **m** and **t** on one of your files.

Joining Contiguous Lines

The **j** command joins the line below the current line with the current line.



Join the line below the current line with the current line.

The **j** command does not accept an address, so the general format for the **j** command is:

j<CR>

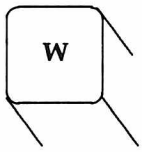
If the current line is not the line you want joined, the easiest way to make it the current line is to display it with **p** or **n**.

1,2p<CR>
Now is the time to join
the team.
p<CR>
the team.
1p<CR>
Now is the time to join
j<CR>
p<CR>
Now is the time to jointhe team.

Notice that there is no space inserted between the last word **join** and the first word of the next line **the**. You will have to place the space between them with the **s** command.

Write Lines of Text to a File

If you are writing the same letter to several different people, you may want to keep the body of the text in a special file to use over again. Those lines of text can be written to the special file with the **w** command.



Write a copy of the contents of the editing buffer to a file.

The general format for the **w** command is:

[address1,address2]w [filename]<CR>

address1,address2
The range of lines to be placed into another file. If you do not use address1 or address2, the entire file is written into a new file.

w The write command.

filename The name of the new file that contains a copy of the block of text.

In the next example the body of the letter is saved in a file called *memo*, so that it can also be sent to other people.

```
1,$n<CR>
1                 March 17, 1985
2           Dear Kelly,
3           There is a meeting in the
4           green room at 4:30 P.M.
5           today.  Refreshments will
6           be served.
3,6w   memo<CR>
87
```

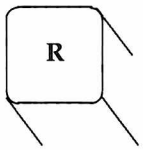
The **w** command has placed a copy of lines three through six into a new file *memo*. **ed** responds to the **w** command with the number of characters in the new file.

Problem:
If there was a file called *memo* in the current directory, it has been erased. The **w** command will overwrite, that is, erase the current file called *memo*, and put the new block of text in the file without giving any warning. In the next section of this tutorial on *Special Commands*, you will learn how to execute shell commands from **ed**. Then, you can list the file names in the directory to make sure that you are not overwriting a file.

Problem:
You cannot write other lines to the file *memo*. If you tried to add lines 13 through 16, the existing lines (3 through 6) would be erased and the file would only contain the new lines 13 through 16.

Read in the Contents of a File

The body of your memo is in a file called *memo*. How do you copy it from that file into the editing buffer?



Read in a copy of the contents of another file into the current editing buffer.

The general format for the **read** command is:

[address1]r filename<CR>

address1 The text will be placed after the line address1. If address1 is not given, the file is added to the end of the buffer.

r The read command.

filename The name of the file that will be copied into the editing buffer.

Using the example from the write command, the next screen depicts editing a new letter and then reading in the contents of the file *memo*.

```
1,$n<CR>
1                    March 17, 1985
2            Dear Michael,
3            Are you free later today?
4            Hope to see you there.
3r memo<CR>
87
3,$n<CR>
3            Are you free later today?
4            There is a meeting in the
5            green room at 4:30 P.M.
6            today.  Refreshments will
7            be served.
8            Hope to see you there.
```

ed responds to the read command with the number of characters in the file *memo* that are now added to the editing buffer.

It is always a good idea to display new or changed lines of text to be sure that they are correct.

SUMMARY OF COMMANDS TO MOVE TEXT

m	Move lines of text.
t	Copy lines of text.
j	Join contiguous lines.
w	Write text into a new file.
r	Read in text from another file.

EXERCISE 6

6-1. There are two ways to copy lines of text in the buffer, one is the copy command, the other is writing the lines of text to a file and then reading the file into the buffer. Writing to a file and then reading the file into the buffer is a longer process. Can you think of an example where this would be more practical? What commands would copy lines 10 through 17 of file *exer* into the file *exer6* at line 7?

6-2. Lines 33 through 46 give an example that you want placed after line 3, and not after line 32. What command performs this task?

6-3. If you are on line 10 of a file and you want to join lines 13 and 14, what commands would you issue?

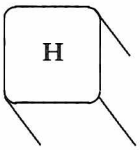
OTHER USEFUL COMMANDS AND INFORMATION

There are four other commands and a special file that will be useful to you when you are editing your files. They are the following:

h,H	The help commands that give error messages.
l	Display characters that are not normally displayed.
f	Display the current file name.
!	Temporarily escape **ed** to execute a shell command.
ed.hup	When a system interrupt occurs, the **ed** buffer is saved in a special file named *ed.hup*.

Help Commands

You may have noticed when you were editing a file that **ed** responds to some of your commands with a ?. The ? is a diagnostic message indicating there is an error. The help commands give you a short message to explain the reason for the most recent diagnostic.



Display a short error message to explain the ? diagnostic.

There are two help commands.

h Display a short error message that explains the reason for the most recent ?.

H Place **ed** in a help mode that displays the short error message each time ? is displayed. The next **H** turns off the help mode.

Let's look at an example of **h** first. At the beginning of this tutorial, you learned that if you tried to quit **ed** without writing the changes in the buffer to a file, you would get a ?. Try it now using **h** to find out what the problem is. When the ? is displayed, type in **h**.

```
q<CR>
?
h<CR>
warning: expecting 'w'
```

The ? is displayed when you give a new file name to the **ed** command line. Examine that ? with **h** to see what the error message is.

```
ed newfile<CR>
? newfile
h<CR>
cannot open input file
```

This error message is telling you there is no file called *newfile*, or if there is a file named *newfile* **ed** is not allowed to read the file.

Now let's examine the **H** command. This command will respond to the ? and then turn on the help mode of **ed**, so that **ed** will give you an explanation each time the ? is displayed until you turn off the help mode with a second **H**. The next screen shows the help mode turned on by **H**. The various error messages are displayed in response to some common mistakes.

```
e newfile<CR>
? newfile
H<CR>
cannot open input file
/hello<CR>
?
search string not found
1,22p<CR>
?
line out of range
a<CR>
This is line one.
.<CR>
s/$ end of line<CR>
?
illegal or missing delimiter
,$s/$/ end of line<CR>
?
unknown command
H<CR>
q<CR>
?
h<CR>
warning expecting 'w'
```

In the preceding example, the help mode is turned on by **H** and displays the error message for ? *newfile*. Then it displays some of the error messages you may encounter in an editing session.

/hello<CR> There is no search pattern **hello** since the buffer is empty.

search string not found

1,22p<CR> There are no lines in the buffer so **ed** cannot print the lines.

line out of range

A line of text is appended to the buffer to show you some error messages associated with the **s** command.

s/$ end of line<CR>
The delimiter between the old text to be replaced and the new text is missing.

illegal or missing delimiter

,$s/$/end of line<CR>
address1 was not typed in before the comma, **ed** does not recognize **,$**.

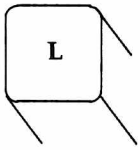
unknown command

The help mode is then turned off and **h** was used to discover the meaning of the last ? . While you are learning **ed**, you may want to leave the help mode turned on so you will use **H**. However, once you become more adept at editing in **ed**, you will only need to see the error message occasionally and so you will use **h**.

Display Nonprinting Characters

If you are typing in a tab character, normally the terminal will display up to eight spaces to the next tab setting. (Your tab setting may be more or less than eight spaces. See *Chapter 7, Shell Tutorial,* on setting **stty-tabs**.)

If you want to see how many tabs you have inserted into your text, you would use the **l** command.



Display nonprinting characters.

The general format for the **l** command is the same as for **n** and **p**.

[address1,address2]l<CR>

address1,address2
The range of lines to be displayed. If no address is given, the current line will be displayed. If only address1 is given, only that line will be displayed.

l The command that displays the nonprinting characters along with the text.

The **l** command denotes tabs with a > character. **l** displays some control characters. These characters are typed in by holding down the CTRL key and pressing another character key. The key that sounds the bell is control g. It is displayed as \07 which is the ASCII hexadecimal representation (the computer's code) for control g.

Type in two lines of text that contain a control g, denoted in the text by **<^g>**, and a tab denoted by **<tab>**. Then use the **l** command to display the lines of text on your terminal as shown below.

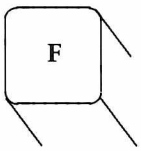
```
a<CR>
Type in <^g> control g.<CR>
Type in a <tab> tab.<CR>
.<CR>
1,2l<CR>
Type in \07 control g
Type in a > tab.
```

Did the bell sound when you typed in **<^g>**?

The Current File Name

In a long editing session, you may forget the file name. The **f** command will remind you which file is currently in the buffer.

Or, you may want to preserve the original file that you entered into the editing buffer and write the contents of the buffer to a new file. In a long editing session, you may forget, and accidentally overwrite the original file with the customary **w** and **q** command sequence. You can prevent this by telling the editor to associate the contents of the buffer with a new file name while you are in the middle of the editing session. This is done with the **f** command and a new file name.



Displays or changes the current file name.

The general format to display the current file name is just **f** alone on a line.

f<CR>

To see how **f** works, enter a file into **ed** and then use the **f** command. The file *oldfile* is entered into **ed** in the example.

```
ed oldfile<CR>
323
f<CR>
oldfile
```

The general format to associate the contents of the editing buffer with a new file name is:

f newfile<CR>

If no file name is given to the write command, **ed** remembers the file name given at the beginning of the editing session and writes to that file. If you do not want to overwrite the original file, you must either use a new file name with the write command, or change the current file name using the **f** command followed by the new file name. Since you can use **f** at any point in the editing session, you can immediately change the currently remembered file name, thus protecting the original file. You can then continue with the editing session without worrying about overwriting the original file.

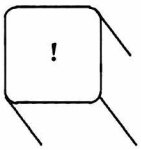
The next screen shows the commands for entering the editor with *oldfile* and then changing the current file name to *newfile*. A line of text is added to the buffer and then the write and quit commands are given.

```
ed oldfile<CR>
323
f<CR>
oldfile
f newfile<CR>
newfile
a<CR>
Add a line of text.<CR>
.<CR>
w<CR>
343
q<CR>
```

Once you have returned to the shell command mode, you can list your files and see that there is a new file named *newfile*. *newfile* should contain a copy of the contents of *oldfile* plus the new line of text.

Escape to the Shell

How can you make sure you are not overwriting an existing file when you write the contents of the editor to a new file name? You need to return to the shell command mode and list your files. The **!** allows you to temporarily return to the shell and execute a shell command line and then return to the current line of the editor.



Temporarily escape to the shell.

The general format for the escape sequence is:

!shell command line<CR>
shell response to the command line
!

When you type in the **!** as the first character on a line, the shell command must follow on that same line. The response to the shell command line will be displayed. When the shell command is finished executing, the **!** will be displayed alone on a line. This tells you that you are back in the editor at the current line.

If you want to return to the shell to find out the correct date, you could type in **!** and the shell command **date**.

p<CR>
This is the current line
! date<CR>
mon Apr 1 14:24:22 CST 1988
!
p<CR>
This is the current line.

The screen first displays the current line. Then, the command is given to temporarily leave the editor and display the date. After the date is displayed, you are returned to the current line of the editor.

If you want to execute more than one command on the shell command line, see the ; in the section on *Special Characters* in *Chapter 7, Shell Tutorial.*

Recover From a System Interrupt

What happens if you are creating text in **ed** and there is an interrupt to the system, you accidentally hung up on the system, or your terminal was unplugged? Is all lost? When there is an interrupt to the system, the UNIX system trys to save the contents of the editing buffer in a special file named *ed.hup*. You can either use the shell command to move *ed.hup* to another file name, or you can put *ed.hup* back into **ed** and use the **f** command to associate the contents of the editing buffer with a new file name. The next screen shows placing *ed.hup* in **ed** and giving it a new file name.

```
ed ed.hup<CR>
928
f myfile<CR>
myfile
```

Conclusion

You now are familiar with many useful commands in **ed**. The commands that were not discussed in this tutorial, such as G, P, Q and the use of **( )** and { }, are discussed in the *Editing Guide*. Their functions are also listed under the **ed** command in the *UNIX System User Reference Manual*. (See *Appendix A*.) You can experiment with these commands and try them out to see what tasks they perform.

SUMMARY OF OTHER USEFUL COMMANDS AND INFORMATION

h	Display a short error message for the preceding diagnostic ?.
H	Turn on the help mode. An error message will be given with each diagnostic ?. The second H turns off the help mode.
l	Display nonprinting characters in the text.
f	Display the current file name.
f newfile	Change the current file name associated with the editing buffer to *newfile*.
! cmd	Temporarily escape to the shell to execute a shell command **cmd**.
ed.hup	The editing buffer is saved in *ed.hup* if the terminal is hung up before a write command.

EXERCISE 7

7-1. Create a new file *newfile1*. Once you have entered **ed**, change the current file name to *current1*. Create some text and write and quit **ed**. If you do the shell command **ls** you will see the directory does not contain a file called *newfile1*.

7-2. Create a file named *file1*. Append some lines of text to the file. Leave the append mode. Do not write the file. Turn off your terminal. Turn on your terminal and log in again. Do an **ls** in the shell. Is there a new file *ed.hup*? Place *ed.hup* in **ed**. How do you change the current file name to *file1*? Display the contents of the file. Are the lines the same lines you created before you turned off your terminal?

7-3. While you are in **ed**, temporarily escape to the shell and send a mail message to yourself.

ANSWERS TO EXERCISES

Exercise 1

1-1.

```
$ ed junk<CR>
? junk
a<CR>
Hello world.<CR>
.<CR>
w<CR>
12
q<CR>
$
```

1-2.

```
$ ed junk<CR>
12
1,$p<CR>
Hello world.<CR>
q<CR>
$
```

The system did not respond with the warning question mark because you did not make any changes to the buffer.

1-3.

```
$ ed junk<CR>
12
a<CR>
This is not Mr. Ed, there is no horsing around<CR>
.<CR>
1,$p<CR>
Hello world.
This is not Mr. Ed, there is no horsing around
q<CR>
?
w stuff<CR>
60
q<CR>
$
```

Exercise 2

2-1.

```
$ ed towns<CR>
? towns
a<CR>
My kind of town is<CR>
Chicago<CR>
Like being no where at all in<CR>
Toledo<CR>
I lost those little town blues in<CR>
New York<CR>
I lost my heart in<CR>
San Francisco<CR>
I lost $$ in<CR>
Las Vegas<CR>
.<CR>
w<CR>
164
```

2-2.

3<CR>
Like being no where at all in

2-3.

−2,+3p<CR>
My kind of town is
Chicago
Like being no where at all in
Toledo
I lost those little town blues in
New York

2-4.

.=<CR>
6
6<CR>
New York

2-5.

$<CR>
Las Vegas

2-6.

?town<CR>
I lost those little town blues in
?<CR>
My kind of town is

2-7.

```
g/in<CR>
My kind of town is
Like being no where at all in
I lost those little town blues in
I lost my heart in
I lost $$ in

v/in<CR>
Chicago
Toledo
New York
San Francisco
Las Vegas
```

Exercise 3

3-1.

```
$ ed ex3<CR>
?ex3
i<CR>
?
q<CR>
```

The ? after the **i** indicates there is an error in the command. There is no current line to insert text before that line.

3-2.

```
$ ed towns<CR>
164
.n<CR>
10          Las Vegas
3i<CR>
Illinois<CR>
.<CR>
.i<CR>
or<CR>
Naperville<CR>
.<CR>
$i<CR>
hotels in<CR>
.<CR>
1,$n<CR>
1           my kind of town is
2           Chicago
3           or
4           Naperville
5           Illinois
6           Like being no where at all in
7           Toledo
8           I lost those little town blues in
9           New York
10          I lost my heart in
11          San Francisco
12          I lost $$ in
13          hotels in
14          Las Vegas
```

3-3.

```
1,5n<CR>
1        My kind of town is
2        Chicago
3        or
4        Naperville
5        Illinois
2,5c<CR>
London<CR>
.<CR>
1,3n<CR>
1        My kind of town is
2        London
3        Like being no where at all
```

3-4.

```
.<CR>
Like being no where at all
/Tol<CR>
Toledo
c<CR>
Peoria<CR>
.<CR>
.<CR>
Peoria
```

3-5.

```
.<CR>
/New Y/c<CR>
Iron City<CR>
.<CR>
.<CR>
Iron City
```

Your search string need not be the entire word or line. It only needs to be unique.

Exercise 4

4-1.

```
v/little/s/town/city<CR>
My kind of city is
London
Like being no where at all in
Peoria
Iron City
I lost my heart in
San Francisco
I lost $$ in
hotels in
Las Vegas
```

The line

I lost those little town blues in

was not printed because it was NOT addressed by the **v** command.

4-2.

```
.<CR>
Las Vegas
s?Las Vegas?Toledo<CR>
Toledo
```

4-3.

```
?lost?s??found<CR>
I found $$ in
```

4-4.

```
/no?s??NO<CR>
?
/no/s//NO<CR>
Like being NO where at all in
```

You can not mix delimiters such as / and ? in a command line.

Exercise 5

5-1.

```
$ ed file1<CR>
? file1
a<CR>
A   Computer Science<CR>
D   Jogging<CR>
C   Tennis<CR>
.<CR>
1,$s/[^AB]/A/<CR>
1,$p<CR>
AA Computer Science
A   Jogging
A   Tennis
u<CR>
```

```
1,$s/^[^AB]/A<CR>
1,$p<CR>
A   Computer Science
A   Jogging
A   Tennis
```

5-2.

```
2i<CR>
These are not really my grades.<CR>
.<CR>
1,$p<CR>
A   Computer Science
These are not really my grades.
A   Tennis
A   Jogging
/^[^A]<CR>
These are not really my grades
?^[T]<CR>
These are not really my grades
```

5-3.

```
1,$p<CR>
I love money
I need money
The IRS wants my money
g/^I/s/I.*m /It's my m<CR>
It's my money
It's my money
```

```
/s/money/gold<CR>
It's my gold
2,$s//%<CR>
The IRS wants my gold
```

5-4.

```
s/10202/&0<CR>
10202031020
```

5-5.

```
a<CR>
* . \ & % ^ *<CR>
.<CR>
s/*/a<CR>
a . \ & % ^ *
s/*/b<CR>
a . \ & % ^ b
```

Because there were no preceding characters, * substituted for itself.

```
s/ \./c<CR>
a c \ & % ^ b
s/ \\/d<CR>
a c d & % ^ b
s/&/e<CR>
a c d e % ^ b
s/%/f<CR>
a c d e f ^ b
```

The & and % are only special characters in the replacement text.

```
s/ \^/g<CR>
a c d e f g b
```

Exercise 6

6-1. Any time you have lines of text that you may want to have repeated several times, it may be easier to write those lines to a file and read in the file at those points in the text.

If you want to copy the lines into other files you must write them to a file and then read in that file into the buffer containing another file.

```
ed exer<CR>
725
10,17 w temp<CR>
210
q<CR>
ed exer6<CR>
305
7r temp<CR>
210
```

The file *temp* can be called any file name.

6-2.

```
33,46m3<CR>
```

6-3.

```
.=<CR>
10
13p<CR>
This is line 13.
j<CR>
.p<CR>
This is line 13 and line 14.
```

Remember the .= will give you the current line.

Exercise 7

7-1.

```
ed newfile1<CR>
? newfile1
f current1<CR>
current1
a<CR>
This is a line of text<CR>
Will it go into newfile1<CR>
or into current1<CR>
.<CR>
w<CR>
66
q<CR>
ls<CR>
bin
current1
rje
```

7-2.

```
ed file1<CR>
? file1
a<CR>
I am adding text to this file.<CR>
Will it show up in ed.hup?<CR>
.<CR>
```

Turn off your terminal.

Log in again.

```
ed ed.hup<CR>
58
f file1<CR>
file1
1,$p<CR>
I am adding text to this file.
Will it show up in ed.hup?
```

7-3.

```
ed file1<CR>
58
! mail mylogin<CR>
You will get mail when<CR>
you are done editing!<CR>
.<CR>
!<CR>
```

Chapter 6

SCREEN EDITOR TUTORIAL (vi)

PAGE

GETTING ACQUAINTED WITH vi 6-1

HOW TO READ THIS TUTORIAL 6-2

GETTING STARTED 6-5
- How to Set Terminal Configuration 6-5
- How to Access vi 6-7
- How to Create Text 6-8
- How to Leave the Append Mode 6-9
- How to Move the Cursor 6-10
- How to Delete Text 6-12
- How to Add Text 6-13
- How to Quit vi 6-14

EXERCISE 1 6-15

POSITIONING THE CURSOR IN THE WINDOW 6-16
- Character Positioning 6-18
  - Positioning the Cursor to the Right or Left 6-18
  - Positioning the Cursor at the End or Beginning of a Line 6-20
- Searching for a Character on a Line 6-21
- Line Positioning 6-23
- Word Positioning 6-25
- Positioning the Cursor by Sentences 6-27
- Positioning the Cursor by Paragraphs 6-29
- Positioning in the Window 6-30

POSITIONING THE CURSOR IN THE FILE 6-34
- Scrolling the Text 6-35
- Go to a Specified Line 6-40
- Line Numbers 6-40
- Search for a Pattern of Characters 6-41

PAGE

EXERCISE 2 6-45

CREATING TEXT 6-46
- Append Text 6-47
- Insert Text 6-47

EXERCISE 3 6-50

DELETING TEXT 6-51
- Delete Commands in the Text Input Mode 6-51
- Undo the Last Command 6-53
- Delete Commands in the Command Mode 6-54
  - Delete Text Objects 6-55

EXERCISE 4 6-60

CHANGING TEXT 6-60
- Replacing Text 6-61
- Substituting Text 6-62
- Changing Text 6-63

CUTTING AND PASTING TEXT ELECTRONICALLY 6-66
- Moving Text 6-66
  - Fixing Typos 6-67
- Copying Text 6-68
- Copying or Moving Text Using Registers 6-69

EXERCISE 5 6-70

SPECIAL COMMANDS 6-71
- Repeating the Last Command 6-71
- Joining Two Lines 6-72
- Typing Nonprinting Characters 6-72
- Clearing and Redrawing the Window 6-73
- Changing Lowercase to Uppercase and Vice Versa 6-73

PAGE

LINE EDITING COMMANDS 6-74
Write Text to a New File 6-75
Finding the Line Number 6-76
Deleting the Rest of the Buffer 6-77
Adding a File to the Buffer 6-77
Making Global Changes 6-77

QUITTING VI 6-80

SPECIAL OPTIONS FOR vi 6-82
Recovering a File Lost by an Interrupt 6-82
Editing Multiple Files 6-83

EXERCISE 6 6-84

CHANGING YOUR ENVIRONMENT 6-85
Setting the Automatic Carriage Return 6-86

ANSWERS TO EXERCISES 6-88
Exercise 1 6-88
Exercise 2 6-89
Exercise 3 6-90
Exercise 4 6-91
Exercise 5 6-92
Exercise 6 6-92

Chapter 6

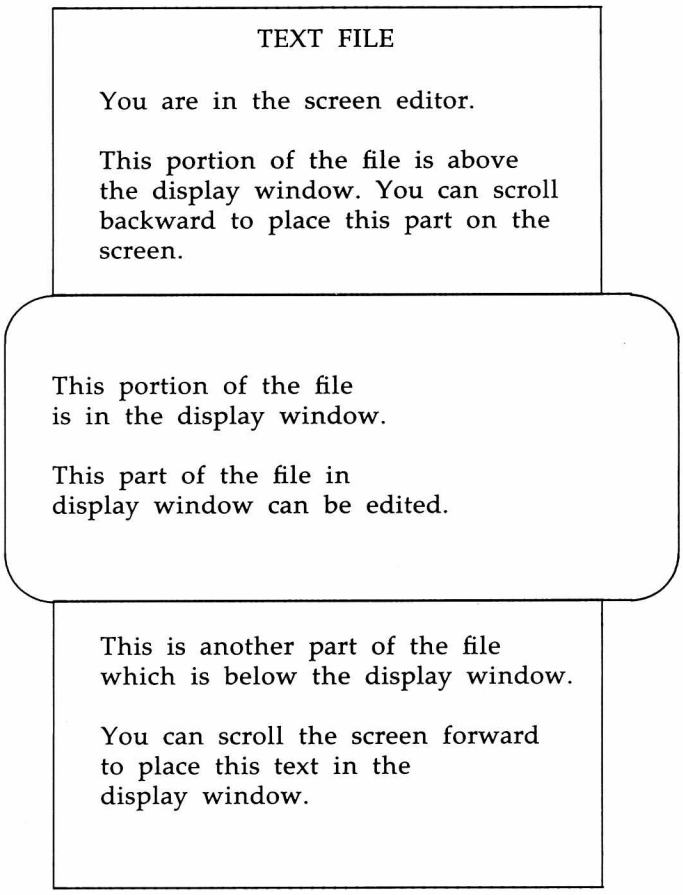
SCREEN EDITOR TUTORIAL (vi)

GETTING ACQUAINTED WITH vi

The screen editor, accessed by the **vi** command, is a powerful and sophisticated tool for creating and editing files. The video display terminal is used as a window to view the text of a file. Within this window, you can add, delete, or change text in much the same way as you would on a typewriter or with paper and pencil. However, making corrections in **vi** does not involve white out, correction tape, or cutting and pasting. A few simple commands change the text, and these changes are quickly reflected in the text on the screen.

The **vi** editor displays from 1 to several lines of text. The cursor can be moved to any point on the screen and text can be created, changed, or deleted from that point. The text in the file can be scrolled forward to reveal the lines below the current window, the window that is on the screen now. Or, the file can be scrolled backward to reveal lines above the current window. (See the display on *page 6-2*.) Other commands can place you at the beginning or end of the file, paragraph, line, or word.

Besides the convenience of editing portions of text within the window, **vi** also gives you the advantage of some line editor commands, such as the powerful global commands that make the same change throughout the whole file.



Editing window of vi displaying part of a file

HOW TO READ THIS TUTORIAL

This chapter is a tutorial on how to access and use **vi**. Although there are more than 100 commands within **vi**, this tutorial covers only the basic commands that will enable you to effectively use **vi**. The following basics will be covered:

- How to set up your particular type of terminal so you can access **vi**,

- How to get started creating a file, deleting some of your mistakes, writing the text into a UNIX system file, and then leaving **vi** to go back to the shell command mode,
- How to move around within the file, so that you can create, delete, or change text,
- How to electronically cut and paste your text,
- How to use some special commands and shortcuts,
- How to temporarily escape to the shell to perform some shell commands and then return to edit the current window of text,
- How to use some line editing commands within **vi**,
- How to quit vi,
- How to edit several files in the same session,
- How to recover a file lost by an interruption to an editing session, and
- How to change your shell environment to automatically set your terminal configuration, and set an automatic carriage return.

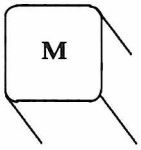
In this tutorial, commands printed in **bold** should be typed into the system exactly as shown. UNIX system responses to those commands are printed in *italic*. The **vi** editor commands that do not print out on the screen will be enclosed in <>. For example, **<CR>** denotes carriage return, meaning press the RETURN key.

The **vi** editor has several commands executed by holding down the "control" or CTRL key while you press another key. These are called control characters. A ^ and a letter denote a control character in the text. For example, **^d** means hold down the control key and press the "d" key. Since **^d** is a command that does not appear on the screen, it will appear in the text as **<^d>**, meaning you should execute **vi** command **<^d>**. As you read the text you may want to glance back for a quick review of these conventions, which are summarized next.

bold command	(Type in exactly as shown.)
italic response	(The system's response to a command.)
roman	(Text that is being typed in a file.)
<CR>	(Commands that are typed in, but not reflected on the screen are enclosed in < >.)
^g	(A control character. Hold down the control key, CTRL, while you press "g".)

In the following sections, a full or partial screen may be used to display the examples showing how the commands are executed. An arrow will point to the letter that is over the cursor. Cursor movements on the screen are depicted by arrows pointing in the direction that the cursor will move.

The keys on your keyboard may be depicted as shown in the example of the "m" key.



Notice that the letter on the key appears as it does on your keyboard. However, when you press the key it will appear in lowercase in your text. If you need an uppercase letter, the example will include the SHIFT key.

The commands discussed in each section are reviewed at the end of the section. A summary of all the **vi** commands is found in *Appendix E,* where they are listed in alphabetical order, as well as by topic.

At the end of some sections, exercises are given for you to experiment with those commands covered in the section. The answers to all of the exercises are at the end of this chapter.

GETTING STARTED

The best way to learn **vi** is to log into the UNIX system and do the examples and the exercises as you read the tutorial. If you experiment with the commands, they will become familiar to you and you will soon be adept at editing in **vi**.

You should be logged into the UNIX system, and ready to create a file in your current directory, the directory you are in now.

How to Set Terminal Configuration

Before you access **vi**, you must set your terminal configuration. That is, you must tell the system what kind of terminal will display the editing window of your file. Each type of terminal has a code name that can be recognized by the system. The code for your terminal is in the UNIX system file */etc/termcap*. The *termcap* file contains information about different terminals. You only need to know the code for your terminal, which is the first two letters of the line containing information about your terminal.

To find the code for your type of terminal, use the **grep** command to search the */etc/termcap* file for your terminal type. For example, if you have a TELETYPE 5420 terminal, type in the following from your login directory:

```
$ grep "teletype 5420" /etc/termcap<CR>
T7|5420|tty5420|teletype 5420 80 columns:
$
```

The code for a Teletype 5420 is T7.

To set the terminal configuration, type in:

TERM=code<CR>
export TERM<CR>

TERM must be typed in uppercase and there are no spaces on either side of the equal sign. "code" will be the first two letters on the line for your terminal from the *termcap* file. In this command sequence, the **export** command assigns the terminal type to your login environment for this session while you are logged in to the UNIX System. You can learn more about exporting variables such as TERM in *Chapter 7, Shell Tutorial* and in *UNIX System Shell Commands and Programming*. (See *Appendix A*.)

In the example below, you have logged into the UNIX system and have gotten your **$** prompt from the system. Then, you set your terminal configuration for the Teletype 5420.

```
$ TERM=T7<CR>
$ export TERM<CR>
$
```

Look up your terminal code in the *termcap* file, or ask your system administrator for the code. If you set your terminal configuration now, you can do the examples as you read the text.

Do not experiment typing in terminal configurations that do not match your terminal, since you may confuse the UNIX system, and you will either have to log off, hang up, or get the help of the system administrator to restore your login environment.

Later in this chapter, you will learn how to set your shell environment so that you do not have to set the terminal configuration each time that you log in to the UNIX system.

How to Access vi

Now you are ready to access vi.

Type in: **vi filename<CR>**

where *filename* is the name of the file you wish to edit, or the name of the file you are about to create.

After you have set your terminal configuration, you want to create a file called *stuff*. For the purpose of this example, **TERM** is set to T7.

```
$ TERM=T7<CR>
$ export TERM<CR>
$ vi stuff<CR>
```

The **vi** command will clear the screen and display the window for the screen editor. It should look like this:

```
~
~
~
~
~
~
~
~
~
"stuff" [new file]
```

The **vi editor window** initially displays some lines of text. In this example there are no lines of text. The screen editor displays a ~ on each line to indicate the file is empty. The cursor is at the beginning

of the file waiting for the first command. In this example, the cursor appears as a short line. Your video display terminal may indicate the cursor by a blinking line or a reverse color block.

Problem:
If you access **vi** and get the following message you have forgotten to set the terminal configuration.

```
$ vi stuff<CR>
I don't know what kind of terminal you are on - all I have is unknown
[Using open mode]
"stuff" [New file]
```

Type in: **:q<CR>**

This returns you to the shell command mode, now you can set your terminal configuration.

How to Create Text

If you have successfully accessed **vi**, you are in the command mode of the screen editor, and **vi** is waiting for your commands. How do you create some text?

- Press the "a" key, **<a>**. Now you are in the append mode of **vi**. You can add text to the file. The **a** does not print out on the screen.
- Start typing in some text.
- To begin a new line press the carriage return key **<CR>**.
- Notice as you get close to the right margin a bell sounds to remind you to press the carriage return. Terminals which do not have a bell, may warn you another way, such as flashing the screen.

It is possible to set the carriage return so that it is automatic; this is discussed later in this chapter in the section on changing your environment.

How to Leave the Append Mode

If you are finished creating text, you need to leave the append mode and return to the command mode of **vi** to edit any text you have created, or to write the text into a UNIX system file. Press the escape key, ESC or DEL, denoted by **<ESC>**. You are now back in the command mode.

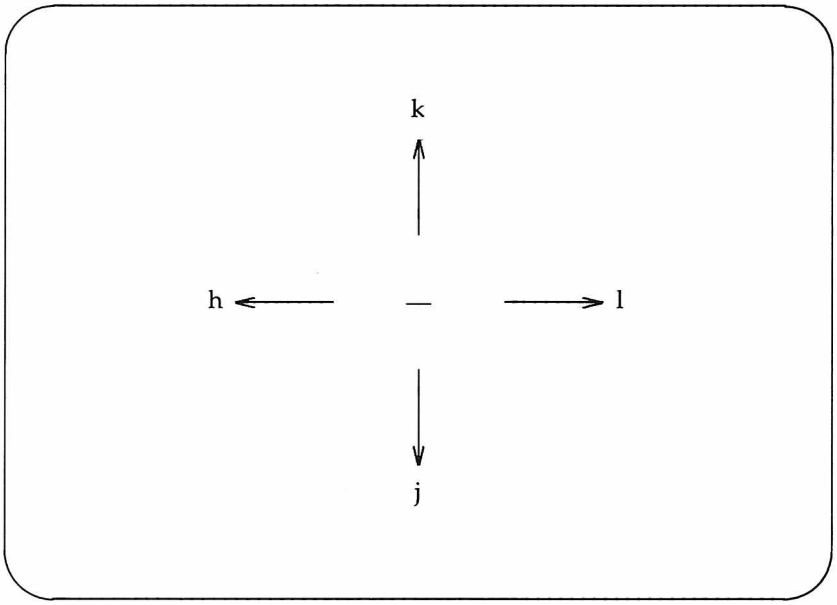
```
<a>
Create some text <CR>
in the screen editor <CR>
and return to the <CR>
command mode. <ESC>
~
~
~
~
~
~
~
~
```

Problem:
If you press **<ESC>** and a bell sounds, **vi** is telling you that you are already in command mode. It will not affect the text in the file if you press **<ESC>** several times. The **vi** editor will only sound a bell each time that you press **<ESC>**.

How to Move the Cursor

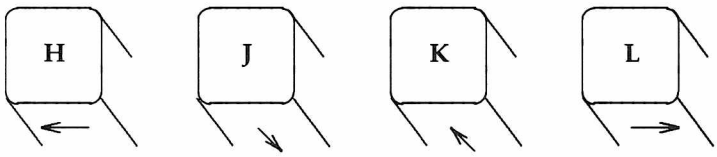
To edit your text, you need to move the cursor to the point on the screen where you will begin the correction. This is easily done with four keys that are next to each other on the keyboard, "h, j, k, l".

<h>	Moves the cursor one character to the left.
<j>	Moves the cursor down one line.
<k>	Moves the cursor up one line.
<l>	Moves the cursor to the right one character.



Right now try moving the cursor around. Watch the cursor on the screen while you press the keys **<h>**, **<j>**, **<k>**, and **<l>**. If you want to move two spaces to the right, press **<l>** twice. If you want to move up four lines, press **<k>** four times. If you cannot go any farther in the direction you have indicated, **vi** will sound a bell.

Many people who use **vi** find it helpful to mark these four keys with arrows indicating the direction that each key moves the cursor. Mark an arrow on each of four small pieces of white correction tape and place a left arrow on the front of the "h" key, a down arrow on the "j" key, an up arrow on the "k" key, and a right arrow on the "l" key.



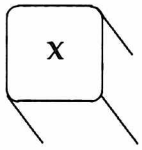
Some terminals have special cursor control keys that are marked with arrows. These may be used as "h, j, k, and l" keys are used.

Problem:
If you are trying to move the cursor around on the screen and the letters h, j, k, and l print out on the screen, you are still in the append mode of **vi**. Press **<ESC>**. Most of the commands in the screen editor are silent, that is they do not print out. If the screen editor commands are printing out on the screen you are still in append mode. Press **<ESC>** and try the commands again.

How to Delete Text

If you have put in an extra character in the text, you will want to delete that character. Move the cursor to that character, and press the "x" key. Watch the screen. The letter will disappear and the line will readjust to the change. If you want to erase three letters in a row, press <x> three times. In the examples below, the position of cursor is depicted by the arrow under the letter.

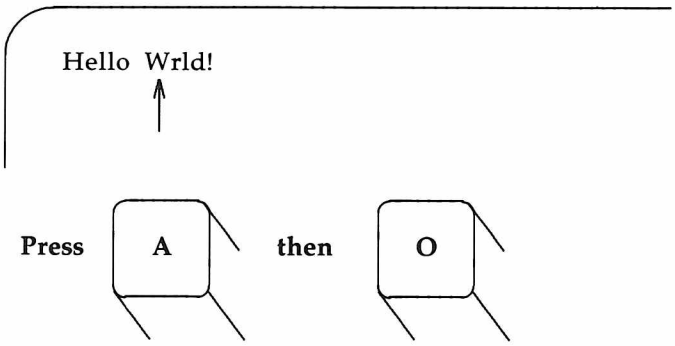
```
Hello Wurld!
      ↑
```

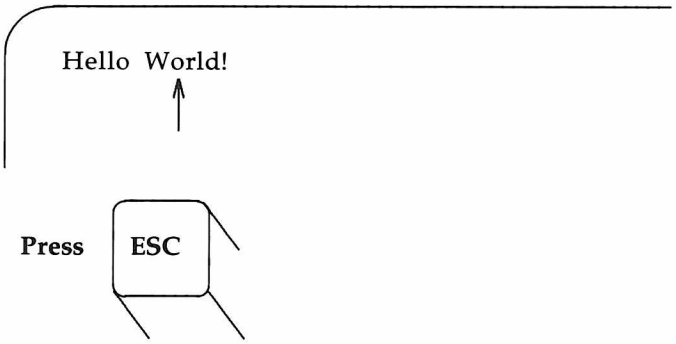
Press 


```
Hello Wrld!
      ↑
```

How to Add Text

If you need to add text at a certain point in the text that is in the window, move the cursor to that point using **<h>**, **<j>**, **<k>**, and **<l>**. Then, press **<a>** and text will be created after that point. As you append text, the characters to the right will move over on the screen to make room for the new characters. The **vi** editor will continue adding all characters that you type in, until you press **<ESC>**. If necessary the characters to the right will even wrap around onto the next line.





Moving around on the screen, or scrolling through the file to add or delete characters, words, or lines, is discussed in detail later in this tutorial.

How to Quit vi

The **vi** command creates a temporary buffer for you. This is equivalent to giving you a piece of scratch paper. When the text or data on the scratch pad is in the form you want for this editing session, you must write it to a UNIX system file. If you are done editing your test file, you will want to put this file in a file called *stuff* in the current directory and get back into the shell command mode.

Hold down the SHIFT key and press the "z" key twice, **<ZZ>**. The **vi** editor remembers the file name given to the **vi** command at the beginning of the editing session, and moves the text from the buffer of the editor to the file named *stuff*. You will get a notice at the bottom of the screen giving the file name, and the number of lines and characters in the file. Then, you are returned to the shell command level, and the UNIX system displays the shell prompt **$**. Since *stuff* is a new file, the notice at the bottom of the screen will include this fact.

```
<a>
This is a test file. <CR>
I am adding text to <CR>
a temporary buffer and <CR>
now it is perfect. <CR>
I want to write this file, <CR>
and return to the shell command <CR>
mode. <ESC><ZZ>
~
~
~
~
"stuff" [New file] 6 lines, 151 characters

$
```

SUMMARY OF GETTING STARTED

TERM=code **export=TERM**	Set the terminal configuration.
vi filename	Enter **vi** editor to edit the file called *filename*.
<a>	Add text after the cursor.
<h>	Move one character to the left.
<j>	Move down one line.
<k>	Move up one line.
<l>	Move to the right one character.
<x>	Delete a character.
<CR>	Carriage return.
<ESC>	Leave the append mode, and return to **vi** command mode.
<ZZ>	Write to a file, and quit **vi**.
:q	Quit **vi**.

EXERCISE 1

There is often more than one way to perform a task in **vi**. If the way you tried worked, then your answer is correct. Watch the screen as you give the commands, and see how it changes or how the cursor moves.

The answers to the exercises are at the end of this chapter.

1-1. If you have not logged in yet, do so now, and set your terminal configuration.

1-2. Enter **vi** and append the following five lines of text to a new file called *exer1*.

This is an exercise!
Up, down
left, right,
build your terminal's
muscles bit by bit.

1-3. Move the cursor to the first line of the file and the seventh character from the right. Notice as you move up the file, the cursor moves "in" to the last letter of the file, but it does not move "out" to the last letter of the next line.

1-4. Delete the seventh and eighth character from the right.

1-5. Move the cursor to the last line of the text, and the last character of that line.

1-6. Append a new line of text.

and byte by byte

1-7. Write the buffer to a file and quit **vi**.

1-8. Reenter **vi** and append two more lines of text to the file *exer1*.

What does the notice at the bottom of the screen say once you have reentered **vi** to edit *exer1*?

POSITIONING THE CURSOR IN THE WINDOW

Until now you have been positioning the cursor with the keys "h, j, k and, l". However, there are several commands to help you move the cursor quickly around the window.

This section on positioning the cursor in the window will look at:

- Positioning by characters on a line,

- Positioning by lines,
- Positioning by text objects
  - — By words,
  - — By sentences, and
  - — By paragraphs, and
- Positioning in the window.

There are also several commands that position the cursor within the **vi** editing buffer. These commands will be looked at in the next section, *Positioning in the File.*

The **vi** editor provides two very helpful patterns in cursor movement.

- Instead of pressing a key such as "h" or "k" a certain number of times, you can precede the command with that number. For example, **<7h>** moves the cursor seven characters to the left.

- Many lowercase commands have an uppercase equivalent that will slightly modify or enhance the command. For example, **<a>** appends text after the cursor, but **<A>** appends text after the last character at the end of the line.

  The uppercase commands will be mentioned briefly in the text, and will be defined in the summary. As you try out the lowercase commands, experiment with the uppercase commands and see what they can do.

If you have not logged into the UNIX system and have not accessed **vi** to edit a file, please do so now. You will want a file that has at least 40 lines in it. If you do not have one, create one now, because you will want to try out each of these cursor movements as you read this section of the tutorial. Remember, to execute these commands, you must be in the command mode of **vi**. Press **<ESC>** to make sure you are out of the append mode, and are in the command mode of **vi**.

Character Positioning

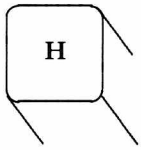
There are three ways to position the cursor by a character on a line.

- You can move the cursor right or left to a character,
- You can specify the character at either end of the line, or
- You can search for a character on a line.

Positioning the Cursor to the Right or Left

The commands, **<h>**, **<l>**, the space bar, and the BACK SPACE key move the cursor right or left to a character on the current line.

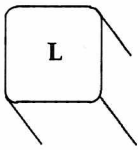
You are already familiar with the "h" and "l" keys.



Move the cursor to the left.

<h> ← Move the cursor one character to the left.

<nh> Move the cursor "n" characters to the left.



Move the cursor to the right.

<l> → Move the cursor one character to the right.

<nl> Move the cursor "n" characters to the right.

Try typing in a number before the command key. Notice that the cursor moves the specified number of characters to the left or right. In the example below, the cursor movement is depicted by the arrows.

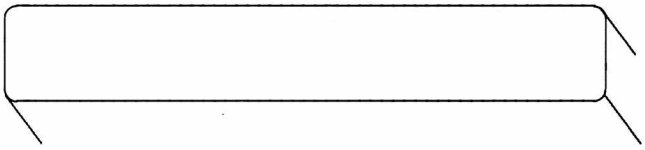
To quickly move the cursor
left or right on the screen,
prefix a number to the command.

Move the cursor left 7 spaces.
← **<7h>**

Move the cursor right three spaces.
<31> →

Even if there are not 100 characters in a line, if you type in **<100l>**, the cursor will simply travel to the end of the line. If you type in **<100h>** the cursor will travel to the beginning of the line.

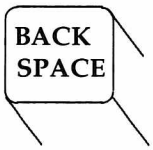
By now, you have probably accidentally discovered that you can move the cursor back and forth on a line using the space bar and the BACK SPACE key.



→ Space bar moves one space to the right

<space bar> → Move the cursor one character to the right.

<nspace bar> Move the cursor "n" characters to the right.



Move the cursor one character to the left.

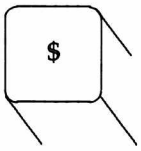
<BS> ← Move the cursor one character to the left.

<nBS> Move the cursor "n" characters to the left.

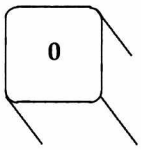
You can type in a number before the space bar or <BS>. The cursor will move that many characters to the left or right.

Positioning the Cursor at the End or Beginning of a Line

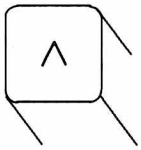
The second method of positioning the cursor on the line is shown below. These commands will place you at the first character or last character of a line.



Position the cursor on the last character of the line.



The number zero positions the cursor on the first character of the line.



The carat key positions the cursor on the first character of the line that is not a blank. (This is not a control character.)

The next examples show the movement of the cursor for each of the three commands.

Go to the back of the line!

<$>

Go to the front of the line!

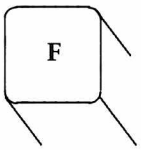
<0> (The number zero)

Go to the first character
of the line that
is not blank!

<^>

Searching for a Character on a Line

The third way to position the cursor on a line is to search for a specific character on the current line. If the character is not on the current line, a bell will sound and the cursor will not move. There is a command that will search the file for patterns. It is discussed in the next section of this tutorial.



Moves the cursor to the right to find the specified letter on the current line.

<fx> → Move the cursor to the right to the specified character *x*.

<Fx> ← Move the cursor to the left to the specified character *x*.

<;> The <;> will continue the search. It will remember the character and seek out the next occurrence of that character on the current line.

In the next example, **vi** is searching to the right for the first occurrence of the letter "A" on the current line.

Go forward to the letter A on this line.

→

<fA>

You may also find the **<tx>** command useful.

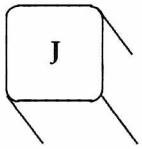
<tx> → Move the cursor to the right, to the character just before the specified character *x*.

<Tx> ← Move the cursor left to the character just after the specified character *x*.

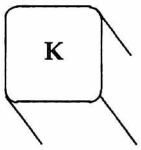
Try the search commands on one of your files. Notice the difference between the uppercase and lowercase commands.

Line Positioning

Besides the **<j>** and **<k>** commands that you have already used, the "+", "−" and RETURN keys will move the cursor line by line. The cursor will try to remain at the same position on the line. If the cursor is on the seventh character from the left in the current line, it will try to go to the seventh character on the new line. If there is no seventh character, the cursor will move to the last character.



↓ **Move the cursor down one line.**



↑ **Move the cursor up on line.**

Since you have already tried out **<j>** and **<k>** and know how they react, try adding a number of lines to the command as you did with **<h>** and **<l>**.

Type in: **7k**

The cursor will move up seven lines above the current line. If there are not seven lines above the current line, a bell will sound and the cursor will remain on the current line.

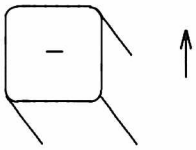
Type in: **35j**

The screen will clear and redraw. The cursor will be on the 35th line below the current line. The new line will be located in the middle of the new window. If there are not 35 lines below the current line, the bell will sound and the cursor will remain on the current line. Try the following command.

Type in: **35k**

Did the screen clear and redraw?

Now, try out the following three easy ways to move up or down in the file.



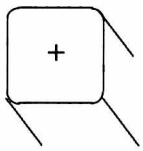
The minus sign moves the cursor up a line.

Type in: **13–**

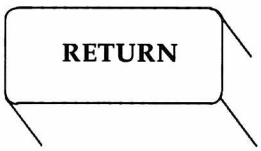
The cursor will travel up 13 lines. If some of those 13 lines are above the current window, the window will move up to reveal those lines. This is a rapid way to move quickly up the file. Try the following command.

Type in: **100–**

What happened to the window? If there are less then 100 lines above the current line, a bell will sound telling you that you have made a mistake, and the cursor will remain on the current line.



or



Move the cursor down a line.

Now, try moving down the lines of the file with +.

Type in: **9+**

The cursor will move down nine lines below the current line.

Try moving down line by line in the file with the RETURN key.

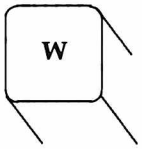
Type in: **5<CR>**

Did the RETURN key give the same response as the "+" key?

Word Positioning

The **vi** editor considers a word a string of characters that are either numbers or letters. The word positioning commands, **<w>**, **<b>**, and **<e>**, consider that any other character is a delimiter, telling **vi** it is the beginning or end of a word. Punctuation before or after a blank is considered a word. The beginning or end of a line is also a delimiter.

The uppercase word positioning commands, **<W>**, **<B>**, and **<E>**, consider that the punctuation is part of the word and define a word by all the characters within two blank spaces, that is, the word is delimited by blanks.



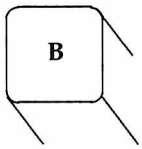
Move the cursor to the right by words.

<w> Move the cursor forward to the first character in the next word. You may press the "w" key as many times as you wish to reach the word you want, or you can prefix the number to the **<w>** command as shown below.

<nw> Move the cursor forward "n" number of words to the first character of that word. The end of the line does not stop the movement of the cursor, it will wrap around and continue to count words from the beginning of the next line.

<W> Ignore all punctuation, and move the cursor forward to the word after the next blank.

The **w** command
leaps word by word through the
file. Move from this word forward
<6w>
six words to this word.



Move the cursor backwards, to the left, by words.

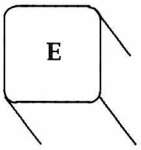
<b> Move the cursor backward one word to the first character of that word.

<nb> Move the cursor backward "n" number of words to the first character of the nth word. The **<b>** command does not stop at the beginning of a line, but moves to the end of the line above and continues to move backward.

<B> Can be used just like the **<b>** command, except that it delimits the word only by blank spaces. It treats all other punctuation as letters of a word.

Leap backward word by word through
the file. Go back four words from here.

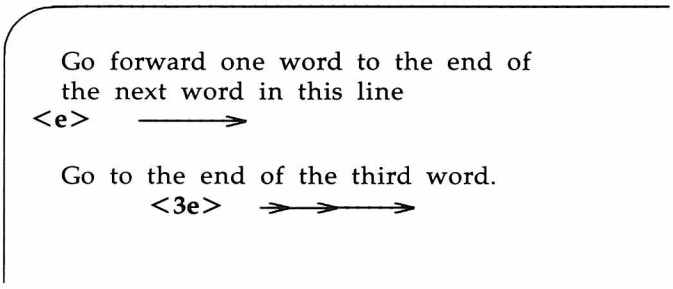
<4b>



Move forward to the end of the word.

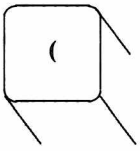
The **<e>** command acts like **<w>** moving forward in the file by words, except that it moves the cursor to the end of the word. This makes it easy to add punctuation or add "s" to the end of a word.

The **<E>** command ignores all punctuation except blanks, delimiting the words only by blanks.

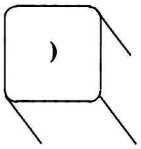


Positioning the Cursor by Sentences

The **vi** editor also recognizes sentences. In **vi**, a sentence ends in "! or . or ?". If they appear in the middle of a line, they must be followed by two blanks spaces for **vi** to recognize them. You should get used to the **vi** convention of putting two spaces at the end of each sentence, because you can also delete, change, or yank whole sentences, which will be discussed later in this tutorial.



Move the cursor to the beginning of a sentence.



Move the cursor to the beginning of the next sentence.

< (> Move the cursor to the beginning of the current sentence.

< n(> Move the cursor to the beginning of the "nth" sentence above the current sentence.

<) > Move the cursor to the beginning of the next sentence.

< n) > Move the cursor to the beginning of the "nth" sentence below the current sentence.

In the next example, the arrows show the movement of the cursor.

This sentence ends in the middle of
a line. Followed by two blank spaces.
← ← <(>

You can go to the end of a sentence.
<)> →

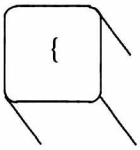
Now, precede the command with a number.

Type in: **3(** or **5)**

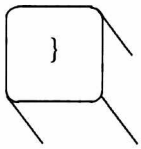
Did the cursor move the correct number of sentences?

Positioning the Cursor by Paragraphs

Paragraphs are recognized by **vi** if they begin after a blank line, or after the paragraph formatting command **.P**. If you want to be able to move the cursor to the beginning of a paragraph (or later in this tutorial, delete or change a whole paragraph), then make sure each paragraph ends in a blank line.



Move the cursor to the beginning of the current paragraph.



Move the cursor to the beginning of the next paragraph.

< { > Move the cursor to the beginning of the current paragraph, which is delimited by a blank line above it.

< n{ > Move the cursor to the beginning of the paragraph, "n" number of paragraphs above the current paragraph.

< } > Move the cursor to the beginning of the next paragraph.

< n} > Move the cursor to the "nth" paragraph below the current line.

The next example uses arrows to show the cursor moving down to the beginning of the paragraph.

The end of a paragraph is
a blank line.

This is a new paragraph.
It also ends in a blank
line. <}>
Go to the beginning
of the next paragraph.

This is the third paragraph.

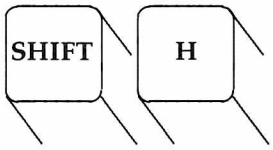
Try moving the cursor with the following commands.

Type in: {
3{
6}

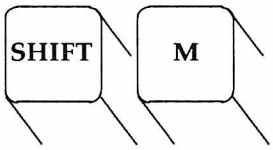
Did you have enough blank lines in your file to test out the last two commands?

Positioning in the Window

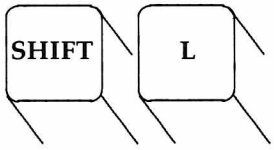
The next three commands help you quickly position yourself in the window. Try out each of the commands.



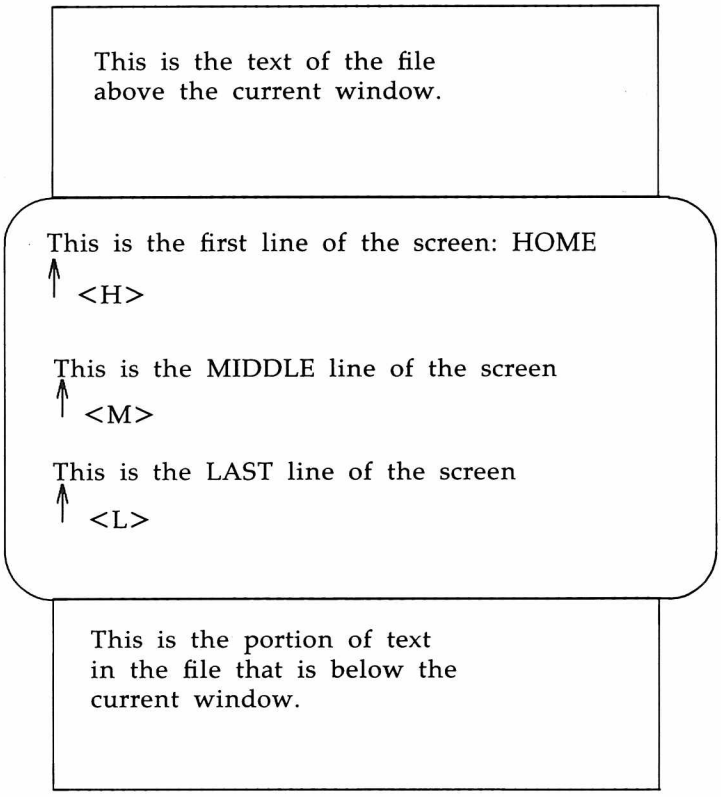
Move the cursor to the first line on the screen.



Move the cursor to the middle line on the screen.



Move the cursor to the last line on the screen.



SUMMARY OF POSITIONING IN THE WINDOW

Character Positioning Commands

<h>	←	Move the cursor one character to the left.
<l>	→	Move the cursor one character to the right.
<BS>	←	Move the cursor one character to the left.
<space bar>	→	Move the cursor one character to the right.
<fx>	→	Move the cursor to the right to the specified character *x*.
<Fx>	←	Move the cursor to the left to the specified character *x*.
<;>		Continue the search. It will remember the character and seek out the next occurrence of the character on the current line.
<tx>	→	Move the cursor to the right, to the character just before the specified character *x*.
<Tx>	←	Move the cursor left to the character just after the specified character *x*.

Positioning by Lines

<j>	Move the cursor down one line in the same column, if possible.

(Continued on next page)

SUMMARY OF POSITIONING IN THE WINDOW *(continued)*

<k>	Move the cursor up one line in the same column, if possible.
<–>	Move the cursor up one line.
<+>	Move the cursor down one line.
<CR>	Move the cursor down one line.

Word Positioning

<w>	Move the cursor forward to the first character in the next word.
<W>	Ignore all punctuation, and move the cursor forward to the next word delimited only by blanks.
<b>	Move the cursor backward one word to the first character of that word.
<B>	Move the cursor to the left one word, which is delimited only by blanks.
<e>	Move the cursor to the end of the current word.
<E>	Delimit the words by blanks only. The cursor is placed on the last character before the next blank space, or end of the line.

(Continued on next page)

SUMMARY OF POSITIONING IN THE WINDOW *(continued)*

Positioning by Sentences

< (>	Move the cursor to the beginning of the current sentence.
<) >	Move the cursor to the beginning of the next sentence.

Positioning by Paragraphs

< { >	Move the cursor to the beginning of the current paragraph.
< } >	Move the cursor to the beginning of the next paragraph.

Positioning in the Window

<H>	Move the cursor to the first line on the screen, or "home".
<M>	Move the cursor to the middle line on the screen.
<L>	Move the cursor to the last line on the screen.

POSITIONING THE CURSOR IN THE FILE

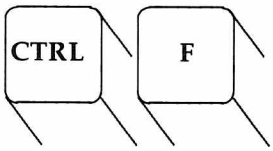
How do you move the cursor to text that is not in the current editing window? You can type in the commands **<20j>** or **<20k>**. However, if you are editing a large file, you need to move quickly

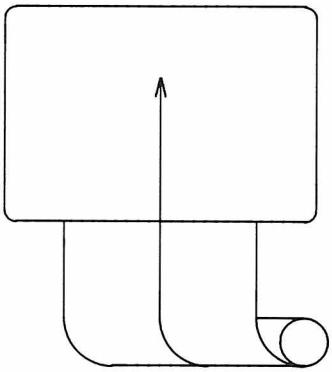
and accurately to another place in the file. This section covers those commands that help you move around within the file. You can:

- Scroll forward or backward in a file,
- Go to a specified line in the file, or
- Search for a pattern in the file.

Scrolling the Text

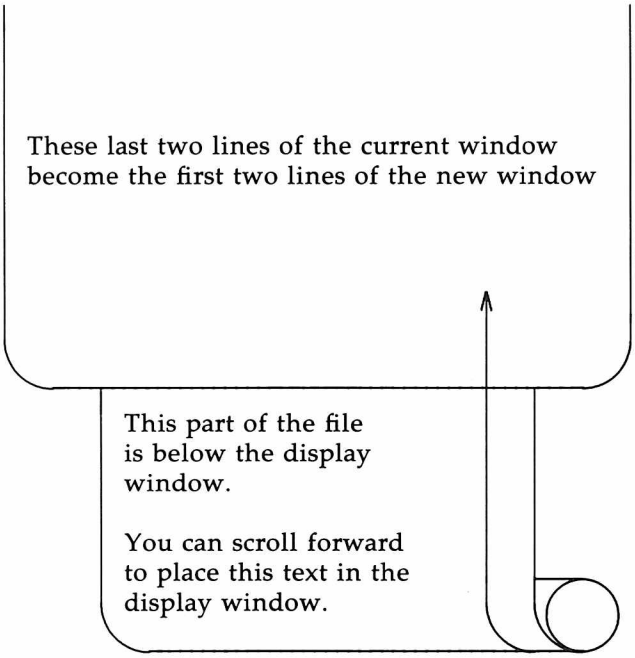
Four basic commands scroll the text of the file. **<^f>** and **<^d>** scroll the screen forward. **<^b>** and **<^u>** scroll the screen backward.



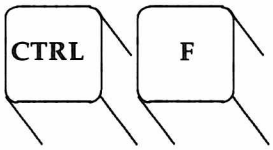


<^f> Scroll the text forward one full window, revealing the window of text below the current window.

To scroll the file forward, **vi** clears the screen and redraws the window. The last two lines that were at the bottom of the current window are placed at the top of the new window. If there are not enough lines left in the file to fill the window, the screen will display the ~ to indicate the empty lines.

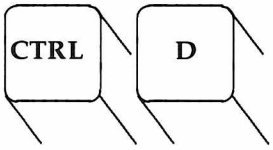


Type in:



vi clears the screen and redraws the new screen shown next.

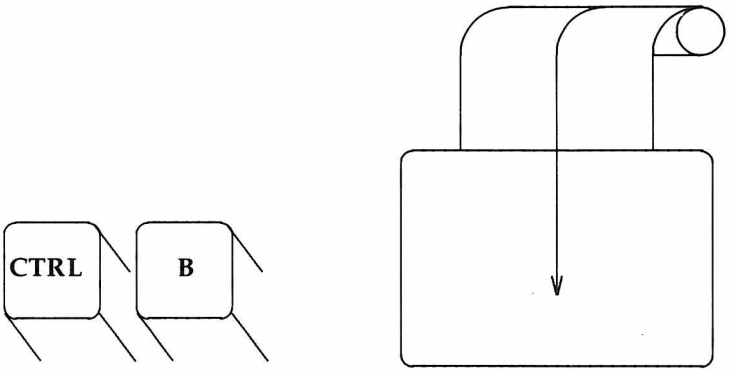
```
These last two lines of the current window
become the first two lines of the new window

This part of the file
is below the display
window.

You can scroll forward
to place this text in the
display window.
~
~
```



Scroll down a half screen to reveal lines below the window.

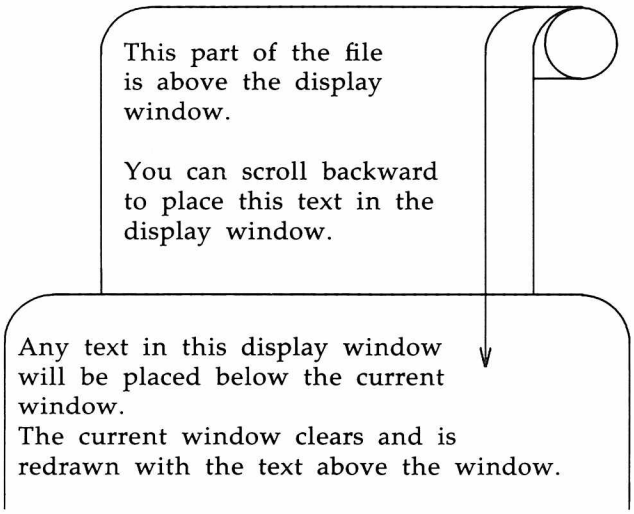
<^d> Scroll down a half screen to reveal text below the window.

When you use **<^d>**, it seems as if the text is being rolled up at the top and unrolling at the bottom to allow the lines below the screen to appear on the screen, while the lines at the top of the screen disappear. If there are not enough lines in the file, a bell will sound indicating there are no more lines.

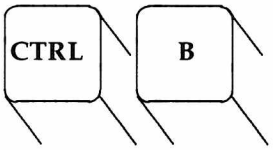


<^b> Scroll the screen back a full window to reveal the text above the current window.

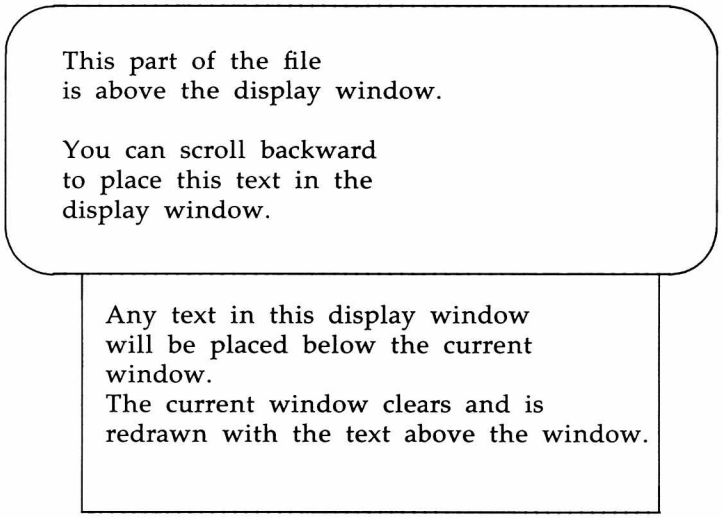
The **<^b>** command clears the screen and redraws the window with the text that is above the current screen. Unlike the **<^f>** command, **<^b>** does not leave any reference lines from the previous window. Also, it does not use the ~ to indicate space above the top of the file. If there are not enough lines above the current window to fill a full new window, a bell will sound and the current window will remain on the screen.



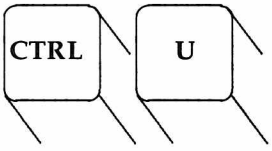
Type in:



vi clears the screen and redraws the new screen shown next.



Any text that was in the display window is placed below the current window.



Scroll up a half screen to reveal lines above the window.

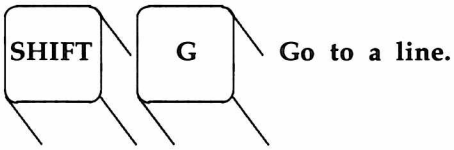
<^u> Scroll up a half window of text to reveal the lines just above the window. At the same time, the lines at the bottom of the window will be erased.

When you use <^u>, it appears as though the text in the file is on a scroll that is being unwound at the top and wound up at the bottom of the screen.

When the cursor is near the top of the file, it will move to the first line of the file and then sound a bell, alerting you it cannot scroll any farther. Try the <^u> and <^d> commands now. Watch the file scroll through the window.

Go to a Specified Line

The <G> command will position the cursor on a specified line in the window, or it will clear the screen and redraw the window around that line. If you do not specify a line, <G> will go to the last line of the file.



Go to a line.

<G>	Go to the last line of the file.
<nG>	Go to the "nth" line of the file.

Line Numbers

Each line of the file has a line number, that corresponds to the number of lines in the buffer. How can you find out the line numbers? There are two basic ways. One way is to use a line editor command, which you will learn about in the section on the line editor commands. The other way is to position the cursor on the line and type in a <^g> command. Try the <^g> command now.

The <^g> command will give you a status notice at the bottom of the screen. The notice tells you:

— Name of the file,

— If the line has been changed [modified],

— Line number,

— Number of the last line in the file, and

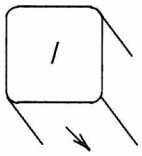
— Percent the current line is of the total lines in the buffer.

```
This line is the 35th line of the buffer.
The cursor is on this line.
                 ↑   <^g>

There are several more lines in the
buffer.
The last line of the buffer is line 116.

"file.name" [modified] line 36 of 116 --34%--
```

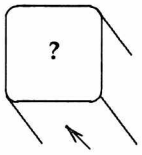
Search for a Pattern of Characters

The fastest way to reach a specific place in your text is to use one of the search commands. You can search forward or backward for the first occurrence of a specified pattern of characters or words in the buffer. The search pattern is ended by **<CR>**.

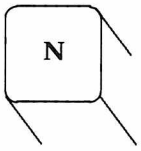
The search commands, / and ?, are not silent. They will print out on the bottom of the screen along with the search pattern. However, the command to repeat the search **<n>** is silent, it does not print out on the bottom of the screen.



Search forward in the buffer.



Search backward in the buffer.



Repeat the previous search.

/pattern<CR>

Search forward in the buffer for the next occurrence of the characters **pattern**. Position the cursor on the first character of the **pattern**.

/Hello world<CR>

Find the next occurrence in the buffer of the two words **Hello world**. Position the cursor under the **H**.

?pattern<CR>

Search backward in the buffer for the first occurrence of the **pattern**. Position the cursor under the first character of the **pattern**.

?data set design<CR>

Search backward in the buffer until the first occurrence of **data set design**. Position the cursor under the "d" of **data**.

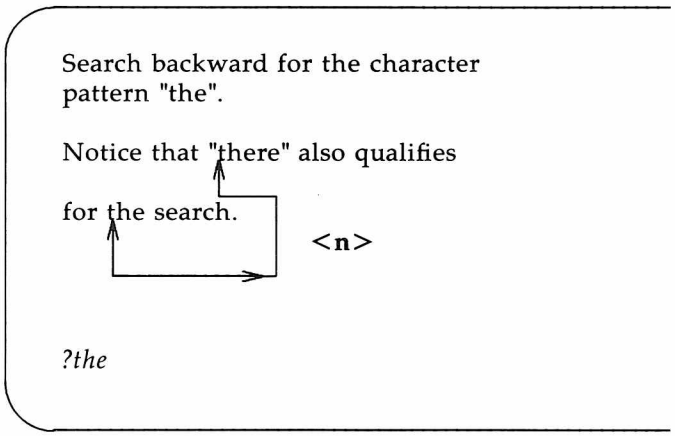
<n> Repeat the last search command.

<N> Repeat the search command in the opposite direction.

The search commands will not wrap around the end of the line in searching for two words. If you are searching for "Hello world", and "Hello" is at the end of one line, and "world" is at the beginning of another line, the search commands will not find that occurrence of "Hello world". However, the search commands will wrap around the end or the beginning of the buffer to continue the search. For example, if you are toward the end of the buffer, and the pattern you are searching for with the **/** command is at the top of the buffer, **/** will find that pattern.

The **<n>** command continues the last search, remembering the pattern and direction of the search.

The following example shows the results of first typing in **?the** and then typing in **<n>**.



Experiment for a minute. What happens if you try to type in a number before **?** or **/** or **<n>**? Experiment with commands in a file called *junk*. If you tried to type in a number before **/** or **?**, you found out it does not work. However, if you tried to type in **<7n>**, you found out that it searched for the seventh identical pattern.

SUMMARY OF POSITIONING IN THE FILE

Scrolling

<^f> Scroll the screen forward a full window, revealing the window of text below the current window.

<^d> Scroll the screen down a half window, revealing lines below the current window.

<^b> Scroll the screen back a full window, revealing the window of text above the current window.

<^u> Scroll the screen up a half window, revealing the lines of text above the current window.

Positioning on a Numbered Line

<G> Go to the last line of the file.

<^g> Give the line number and status.

Searching for a Pattern

/pattern Search forward in the buffer for the next occurrence of the **pattern**. Position the cursor on the first character of the **pattern**.

(Continued on next page)

SUMMARY OF POSITIONING IN THE FILE *(continued)*

?pattern	Search backward in the buffer for the first occurrence of the **pattern**. Position the cursor under the first character of the **pattern**.
<n>	Repeat the last search command.
<N>	Repeat the search command in the opposite direction.

EXERCISE 2

2-1. Create a file called *exer2*. Type a number on each line, numbering the lines from 1 to 50. Your files should look similar to the following.

```
1
2
3
4
5
.
.
.
45
46
47
48
49
50
```

2-2. Try using each of the scroll commands, notice how many lines scroll through the window. Try the following:

<^f>
<^b>
<^u>
<^d>

2-3. Go to the end of the file. Append the following line of text.

123456789 123456789

What number does the command **7h** place the cursor on? What number does the command **3l** place the cursor on?

2-4. Try the command **$** and the command **0** (number zero)

2-5. Go to the first character on the line that is not a blank. Move to the first character in the next word. Move back to the first character of the word to the left. Move to the end of the word.

2-6. Go to the first line of the file. Try the commands that place the cursor on the middle of the window, on the last line of the window, and on the first line of the window.

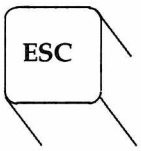
2-7. Search for the number 8. Find the next occurrence of number 8. Find 48.

CREATING TEXT

There are three basic commands for creating text:

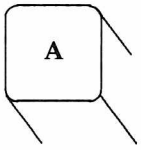
- Append command **<a>**,
- Insert command **<i>**, and
- Open command that creates text on a new line **<o>**.

After you finish creating text with any one of these commands, you can return to the command mode of **vi** with the **<ESC>** command.



The ESC key ends the text input mode.

Append Text



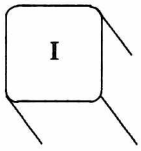
Append text.

<a> Create text to the right of the cursor, or after the cursor.

<A> Append text at the end of the current line.

You have already experimented with the <a> command in the section on *Getting Started*. Make a new file named *junk2*. Append some text using the <a> command. Escape or return to the command mode of **vi** by pressing the ESC key. Then, compare the <a> command with the <A> command.

Insert Text



Insert text.

<i> Insert text to the left of the cursor, or before the cursor.

<I> Create text at the beginning of the current line before the first character that is not a blank.

In the example below, the arrow shows where the new text will be created.

Insert before the H of Here.
Insert before the H of → Here.

<i>

Press **ESC**

To end the insert mode and return to the command mode of **vi**, press the "ESC" key. In the next example you can compare the append command with the insert command.

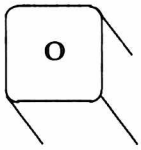
Append after the H of Here.
Append after the H of H → ere.

<a>

Insert before the H of Here.
Insert before the H of → Here.

<i>

Remember to end the append mode and the insert mode with the <ESC> command.



Create a new line of text.

<o> The open command **<o>** creates text at the beginning of a new line below the current line. The cursor can be on any character in the current line.

<O> To create text at the beginning of a new line above the current line, use the **<O>** command.

In the next screen the **<o>** command opens a new line below the current line and begins creating text at the beginning of the new line.

```
Create text with the open line command.

Create text below  the current line.

                <o>
```

SUMMARY OF CREATE COMMANDS

<a>	Create text after the cursor.
<A>	Create text at the end of the current line.
<i>	Create text in front of the cursor.
<I>	Create text before the first character on the current line that is not a blank.
<o>	Create text at the beginning of a new line below the current line.
<O>	Create text at the beginning of a new line above the current line.
<ESC>	Return **vi** to the command mode from any of the above text input modes.

EXERCISE 3

3-1. Create a test file *exer3*.

3-2. Insert the following four lines of text.

Append text
Insert text
a computer's
job is boring.

3-3. Create a line of text

financial statement and

above the last line.

3-4. Create a line of text

Delete text

above the third line using an insert command.

3-5. Create a line of text

byte of the budget

below the current line.

3-6. Using an append command create a line of text

But, it is an exciting machine.

below the last line.

3-7. Move to the first line and append "some" before "text".

Now, practice each of the six commands for creating text until you are familiar with using them.

3-8. Leave **vi** and go on to the next section to find out how to delete any mistakes you made in creating text.

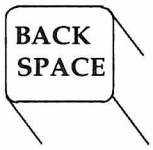
DELETING TEXT

You can delete text from the text input mode or the command mode of **vi**. In addition, you can undo the effect of your most recent command that changed the buffer.

Delete Commands in the Text Input Mode

To delete text in the text input mode, you will use **<BS>**.

<BS> Delete the current character, the character indicated by the cursor.



Delete a character in the create mode of vi.

The BACK SPACE key **<BS>** backs up the cursor in the create mode and deletes each character that the cursor backs across. However, the deleted characters are not erased from the screen until you type over them, or use **<ESC>** and return to the command mode of **vi**.

In the next examples, the arrows show the movement of the cursor.

Press **BACK SPACE** **three times.**

```
<a>
Back space 3 spaces
                  ←
```

Press **ESC**

```
<a>
Back space 3 spa
                ←
```

Notice that the characters do not erase from the screen until you press the ESC key.

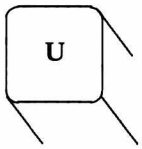
There are two other commands that delete text in the text input mode. Although you may not use them often, you want to be aware that they are commands in the text input mode and need a special command to type them into your text, see the section on special commands.

<^w> Delete the current word, or a specified portion of the word from the cursor to the end of the word.

<@> Delete all of the portion of the line that is currently being created.

Undo the Last Command

Before you experiment with the commands that can delete a good portion of your text, you will want to try out the "undo" command, which will undo the last command.



Undo the last command.

<u> Undo the last command.

<U> Erase the last change on the current line.

If you deleted a line, **<u>** will bring it back on the screen. If you hit the wrong command, **<u>** will undo that command.

If you press the "u" key twice, it will undo the "undo". That is, if you delete a line, the first **<u>** will restore the line. If you press **<u>** again, it will delete the line again.

Delete Commands in the Command Mode

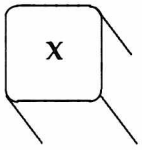
You know that you can precede a number before the command. Many of the commands in **vi**, such as the delete and change commands, allow an argument after the command. The argument can specify a text object such as a word, or a line, or a sentence, or a paragraph. The general form of a **vi** command is:

[number]command[argument]

The brackets around objects in the general form of the command line denote optional parts of the command. They are not part of the command line.

You will see many examples of this form for the delete and change commands.

All of the delete commands in the command mode of **vi** immediately remove the deleted text from the screen and redraw that part of the screen.



Delete a character.

<x> Delete one character.

<nx> Delete "n" characters, where n is the number of characters you want to delete.

You used <x> in the *Getting Started* section of this chapter. Now try preceding <x> with the number of characters you want to delete.

```
Tomorrow the Loch Ness monster
shall slither forth from the
deep dark deep depths of the lake.
```

Put the cursor on the first letter you want to delete, in this example the "d" of the second "deep".

Type in: **5x**

The screen will delete "deep", plus the extra space, and readjust the text on the screen so that it will now read:

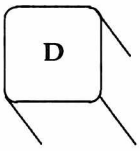
```
Tomorrow the Loch Ness monster
shall slither forth from the
deep dark depths of the lake.
```

You can also use the delete word command, which is discussed next.

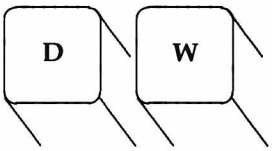
Delete Text Objects

The delete command follows the general form of a **vi** command.

[number]d[text object]



Delete a word, a line, a sentence, or a paragraph.



Delete a word.

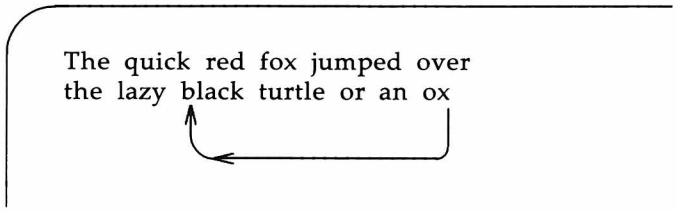
You can delete all of a word or part of a word with **<dw>** by moving the cursor to the first character you want deleted. Pressing **<dw>** deletes that character and all characters up to and including the next space or punctuation character.

```
To delete part of thisill word.
                   ↑
```

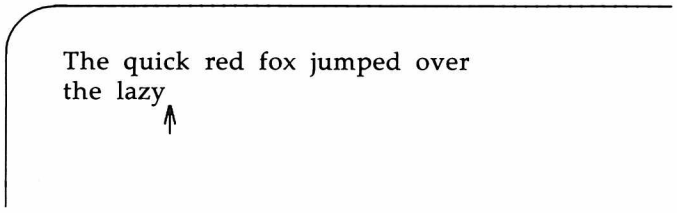
Type in: **dw**

```
To delete part of thisword.
                   ↑
```

You can delete one word with **<dw>** or several words by prefixing the "dw" with a number. The cursor must be on the first character of the first word to be deleted. To delete five words, you would type in **5dw**. An example of how to do this follows.



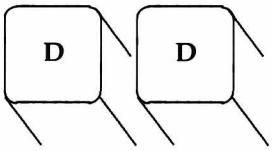
Type in: **5dw**



Try typing in the arguments for other text objects that you learned in the section on positioning the cursor.

Type in: **d(** or **d}**

Observe what happens to your file. Remember, you can restore the text that you just deleted with **<u>**.



<dd> Delete a line of text.

To delete a line, press the "d" key twice. You do not need to worry about deleting text if you press the "d" key once. Nothing will

happen, unless you press the space bar. The **<d space bar>** acts like the **<x>** command and deletes one character. If you accidentally press "d" key in the command mode, press the ESC key. The ESC key will cancel the previous typed command.

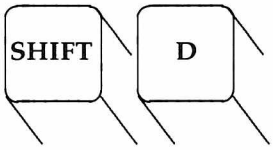
Try to delete ten lines.

Type in: **10dd**

The lines will be deleted from the screen. If some of the lines are below the current window, **vi** will display a notice on the bottom of the screen:

10 lines deleted

If there are not ten lines below the current line in the file, a bell will sound and no lines will be deleted.



Delete the line from the cursor to the end of the line.

If you are erasing the end of a line, use the **<D>** command. Put the cursor on the first character to be deleted, hold down the SHIFT key while you press the "d" key.

Type in: **D**

The **<D>** command will not allow you to specify more than the current line. You cannot type in "3D". However, you could type in **<3d$>**. Remember the general form of a **vi** command? The **$** refers to the end of the line in **vi**.

SUMMARY OF DELETE COMMANDS

For the CREATE Mode:

<BS>	Delete the current character.
<^h>	Delete the current character.
<^W>	Delete the current word.
<@>	Delete the current line of new text, or delete all new text on the current line.

For the COMMAND Mode:

<u>	Undo the last command.
<U>	Erase the last change on the current line.
<x>	Delete the current character.
<n**d**x>	Delete "n" number of text objects "x".
<dw>	Delete the word at cursor through the next space or to the next punctuation mark.
<dd>	Delete the current line.
<D>	Delete the line at the cursor to the end of the line.
<d)>	Delete the current sentence.
<d}>	Delete the current paragraph.

EXERCISE 4

4-1. Create a file *exer4* containing the following four lines:

> When in the course of human events
> there are many repetitive, boring
> chores, then one ought to get a
> robot to perform those chores.

4-2. Move the cursor to line 2 and append to the end of that line:

> tedious and unsavory.

Delete "unsavory" while in the append mode.

Delete "boring" in the command mode.

What is another way you could have deleted "boring"?

4-3. Insert at the beginning of line 4:

> congenial and computerized.

Delete the line.

How could you delete the line and leave it blank?

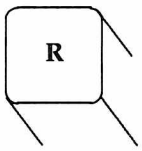
Delete all the lines with one command.

4-4. Leave the screen editor and remove the empty file from your directory.

CHANGING TEXT

Instead of deleting text using a delete command and then creating text with a text input command, the three basic commands, **<r>**, **<s>**, and **<c>** both erase the text and then create new text.

Replacing Text



Replace one character that is typed over.

<r> Replace the current character, the character pointed to by the cursor. This is not a text input mode. It does not need to be ended by <ESC>.

<nr> Replace "n" characters with the same letter. This command automatically terminates after "nth" character is replaced. It does not need the <ESC>.

<R> Replace only those characters typed over until the <ESC> command is given. If the end of the line is reached, this command will then begin appending new text.

The <r> command will replace the current character with the next character that is typed in. For example, in the sentence below you want to change "acts" to "ants".

The circus has many acts.

Place the cursor under the "c" of "acts".

Type in: **rn**

The sentence becomes:

The circus has many ants.

To change "many" to "6666", place the cursor under the "m" of "many".

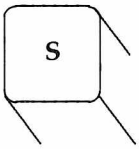
Type in: **4r6**

The <r> command changes the four letters of "many" to 6s.

The circus has 6666 ants.

Substituting Text

The substitute command replaces characters, but then allows you to continue to create text from that point until you press <**ESC**>.



Substitute for a character of text.

<s> Delete the character the cursor is on and append text. End the text input mode with the ESC key.

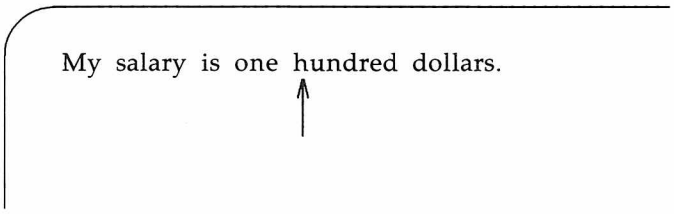
<ns> Delete "n" characters and append text. End the text input mode with <**ESC**>.

<S> Replace all the characters in the line.

The <s> command indicates the last character in the substitution with a $. The characters are not erased from the screen until you type over them, or leave the text input mode with the <**ESC**> command.

Notice that you cannot use an argument with either <r> or <s>. Did you try?

Suppose you want to substitute "million" for "hundred" in the following example.



Put the cursor under the h of hundred.

Then type in: **7s**

Notice where the **$** is placed.

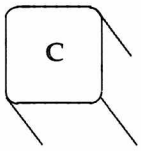
My salary is one hundre$ dollars.

Now type in: **million**

Press the ESC key, and you will owe the Internal Revenue Service $500,000.

Changing Text

The substitute command replaces characters. The change command replaces text objects, and then continues to append text from that point until you press **<ESC>**. To end the change command and return to the command mode in **vi**, you must press the ESC key.



Change. Replace a text object with new text.

The change command can take an argument. You can replace a character, word, or an entire line with new text.

<cw> Replace a word or the remaining characters in a word with new text. The **vi** editor prints a **$** indicating the last character to be changed.

<ncw> Replace "n" number of words with new text.

<cc> Replace all the characters in the line.

<ncc> Replace all the characters in the current line and up to "n" lines of text.

<ncx> Replace "n" number of text objects "x", such as sentences) and paragraphs }.

<C> Replace the remaining characters in the line, from the cursor to the end of the line.

<nC> Replace the remaining characters from the cursor in the current line and replace all the lines under the current line up to "n" lines.

For the **<cw>** command and the **<C>**, a **$** will indicate the last letter that will be replaced. The characters will remain on the screen until you have pressed the ESC key. When used to change one or more lines of text, the change command simply deletes the lines that are to be replaced, and then places you in the text input mode of **vi**.

To change a word, use the **<cw>** command. In the next line change the word "chang$" to "replace".

<cw>

In the example, notice that "replace" has more letters then "change". Once you have executed the change command you are in the text input mode of **vi** and you can add as much text as you want, until you press <ESC>.

To change a word, use the **<cw>** command. In the next line change the word "replace" to "replace".

↑

<ESC>

Try the other change commands. Watch the screen. When you use **<C>** the **$** will appear at the end of the line. Try using other arguments.

Type in: **c{**

Since you know the undo command, do not hesitate to experiment with different arguments, or preceding the command with a number. You must press **<ESC>** before you can use **<u>** since **<c>** places you in a text input mode.

Compare **<S>** to **<cc>**. The results should be the same for both commands.

SUMMARY OF CHANGE COMMANDS

<r>	Replace only the current character.
<R>	Replace only those characters typed over with new characters until the **<ESC>** command is given.
<s>	Delete the character the cursor is on and append text. End the append mode with the ESC key.
<S>	Replace all the characters in the line.

(Continued on next page)

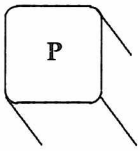
SUMMARY OF CHANGE COMMANDS *(continued)*	
<cw>	Replace a word or the remaining characters in a word with new text.
<cc>	Replace all the characters in the line.
<ncx>	Replace "n" number of text objects "x", such as sentences) and paragraphs }.
<C>	Replace the remaining characters in the line, from the cursor to the end of the line.

CUTTING AND PASTING TEXT ELECTRONICALLY

There is a set of commands that will cut and paste text in a file. Another set of commands will copy a portion of text and place it in another section of a file.

Moving Text

You can move text from one place to another in the **vi** buffer by deleting the lines and then placing them at the spot in the text that you want them. The last text or lines that were deleted go into a temporary buffer. If you move the cursor to that part of the file where you want the deleted lines to be placed and press the "p" key, the deleted lines will be added below the current line.



The put command <p> puts the last yank or delete in the proper place.

<p> Place the contents of the temporary buffer after the cursor.

<np> Place "n" number of copies of the temporary buffer after the cursor.

A partial sentence that was deleted by the <D> command can be placed in the middle of another line. Position the cursor in the space between two words, then press "p". The partial line is placed after the cursor.

Characters deleted by <nx> also go into a temporary buffer. Any text object that was just deleted can be placed somewhere else in the text with <p>.

The <p> command should be used right after a delete command since the temporary buffer only stores the results of one command at a time. The <p> command also places a copy of text after the cursor that had been placed in the temporary buffer by the yank command. Yank <y> is discussed next in *Copying Text*.

Fixing Typos

A quick way to fix typos that consist of transposed letters is to combine the <x> and the <p> commands as <xp>. <x> deletes the letter. <p> places it after next character.

Notice the error in the next line.

A line of tetx

This error can be quickly changed by placing the cursor under the "t" in "tx" and then pressing first "x" and then "p" keys. The result is:

A line of text

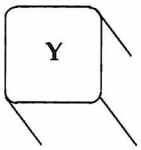
Try it. Make a typing error in your file. Then use <xp>.

Copying Text

You can "yank" (copy) a part of the text into a temporary buffer, then move the cursor to that part of the file where you want to place a copy of the text, and place it there. **<p>** places the text after the current line.

The "yank" command follows the general form of a **vi** command. It allows you to specify the number of text objects that you want copied.

[number]y[text object]



The "yank" command <y> saves a copy of the text object.

<yw>	Yank a copy of a word.
<yy>	Yank a copy of the current line into a temporary buffer to be placed below another line.
<nyy>	Yank "n" lines into a temporary buffer to be placed below the current line. "n" is the number of lines.
<y)>	Yank a copy of a sentence.
<y}>	Yank a copy of the paragraph.
<nyx>	Yank "n" number of text objects "x", such as sentences) and paragraphs }.

Try the following command lines and see what happened to your screen. Of course you can undo the last command.

Type in: **5yw**

Move the cursor to another spot.

Type in: **p**

Try yanking a paragraph <y}> and placing it after the current paragraph, then move to the end of the file <G> and place that same paragraph at the end of the file.

Copying or Moving Text Using Registers

If you have several sections of text that you wish moved or copied to a different part of the file, it would be tedious to move each portion one at a time. **vi** has named registers, which are electronic storage boxes where you can store the text until you want to place it into a specific spot in the file. These registers are named for each letter of the alphabet, a through z. You can either yank or delete text to one of these registers.

These commands are handy if you have an example that you want to use several times in the text. The example will stay in the specified register until you end the editing session or yank or delete another section of text to that register.

The general form of the command is:

[number"1]command[text object]

The l represents any letter, and is the name of the register. You can precede the command with a number to indicate how many text objects, such as words or lines, that you want to save in the register.

Place the cursor at the beginning of a line.

Type in: **3"ayy**

Now, type in more text. Then, go to the end of the file.

Type in: **"ap**

Did the lines you saved in register "a" appear at the end of the file?

SUMMARY OF CUT AND PASTE COMMANDS

<p>	Place the contents of the temporary buffer containing the last delete or yank command into the text after the cursor.
<yy>	Yank a line of text and place it into a temporary buffer.
<nyx>	Yank a copy of "n" number of text objects "x" and place them in a temporary buffer.
<"lyn>	Place a copy of text object "n" in the register named by a letter "l".
<"lp>	Place the contents of register l after the cursor.

EXERCISE 5

5-1. Edit the file *exer2*. Notice that this is the same file you created in Exercise 2.

Go to line 8 and change that line to read "END OF FILE".

5-2. Yank the first eight lines of the file and place them in register "z". Put the contents of register "z" after the last line of the file.

5-3. Go to line 8 and change that line to read "8 is great".

5-4. Go to line 18 and make the same change as you did in 5-3.

5-5. Go to the last line of the file. Substitute "EXERCISE" for "FILE". Replace "OF" with "TO".

SPECIAL COMMANDS

There are some special commands that you will find useful.

<.> Repeat the last command.

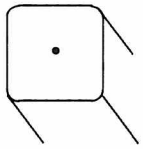
<J> Join two lines together.

<\>
or
<^v> Print out nonprinting character.

<^l> Clear the screen and redraw it.

<~> Change lowercase to uppercase and vice versa.

Repeating the Last Command



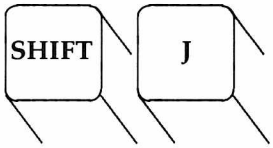
Repeat the last change command.

You may have already accidentally pressed the "." key, thinking that you were adding a period at the end of your sentence. If you were in the command mode of **vi**, you were unpleasantly surprised by the last text change suddenly appearing on the screen.

The period repeats the last change command. This is a very handy command when it is used with the search command. For example, you forgot to capitalize the "S" in United States. However, you do not want to capitalize the "s" in "chemical states". One way you could correct this problem is search for "states". The first time you found "states" in United states, you would change the "s" to "S". The next occurrence you found, you would simply press the "." key and **vi** would remember to change the "s" to "S".

The <.> will repeat change, or create, or delete, or put commands. Experiment with the commands. Watch the screen to see how the text is affected.

Joining Two Lines



Join the line below the current line with the current line.

The <**J**> command joins lines. Place the cursor on the current line, hold down the SHIFT key and press the "j" key. The line below the current line is joined to the current line at the end of the current line.

Now is the time to join
forces.

To join these two lines into one line, place the cursor under any character in the first line.

Type in: **J**

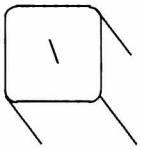
Those two lines become:

Now is the time to join forces.

Notice that **vi** automatically places a space between the last word on the first line and the first word on the second line.

Typing Nonprinting Characters

In the section of this tutorial on deleting in the text input mode, two commands were mentioned that are probably seldom used, but act as commands and will not print out in your text. How do you get characters that are commands in the text input mode to type out in your text? Precede them with a \ .

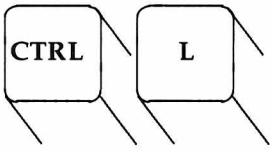


Type in nonprinting characters.

What happens when you want to type in the @ character? Try it. It erased the line you are working on. How do you type in the @ character?

Type in: \@

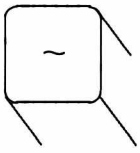
Clearing and Redrawing the Window



Clear and redraw the current screen.

One of the frustrating things that can happen to you in **vi** is that another user in your UNIX system decides to send you a message using the **write** command. If you have not turned off your messages in the shell, the message will appear right at the spot where you are editing in the current window. After you have read the message, how do you restore the current window? If you are in the text input mode, you must end it with the **<ESC>** command to get you into the command mode of **vi**. Then, hold down the CTRL key and press the "l" key. **vi** will clear away the garbage, and redraw the window exactly as it was before the message arrived.

Changing Lowercase to Uppercase and Vice Versa



Change uppercase to lowercase, or lowercase to uppercase.

A quick way to change any lowercase letter to a capital letter or any capital letter to lowercase is the <~> command. To change a to A, or B to b press ~. This command does not allow you type in a number before the command and change several letters with one command.

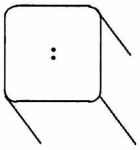
SUMMARY OF SPECIAL COMMANDS

<.>	Repeat the last command.
<J>	Join the line below the current line with the current line.
<\x>	Print the nonprinting character x that does not print out in the text input mode.
<^v>	Print characters that do not normally print out in the text input mode.
<^l>	Clear and redraw the current window.
<~>	Change lowercase to uppercase, or vice versa.

LINE EDITING COMMANDS

The screen editor **vi** also has some line editing capabilities. The line editor associated with **vi** is called **ex**. However, the **ex** commands are very similar to the **ed** commands discussed in *Chapter 5*. If you know the **ed** commands, you may want to experiment on a test file and see how many will work in **vi**.

There are many commands in the **ex** editor that can be called from **vi**. These commands are discussed at length in the *UNIX System Editing Guide*. (See *Appendix A*.) Only a few of the most useful commands are discussed here.



Call in the line editor commands.

To call in the line editor commands, type in a ":" from the command mode of **vi**. The cursor will drop down to the bottom of the screen and display the ":". As you try out the line editing commands notice that they print out at the bottom of the editing window.

A powerful and useful command of **ex** is the command that temporarily returns you to the shell. You can return to the shell, perform some shell commands (even edit and write another file in **vi**) and then return to the current window of **vi**.

:sh<CR> Temporarily return to the shell, leaving the **vi** buffer with the cursor on the current line.

<^d> After you have executed the shell commands, hold CTRL and press "d". You will return to the exact line and window you were editing before you left **vi**.

Even if you change directories while you are temporarily in the shell and then execute **<^d>**, you will return to the **vi** buffer in the directory where you were editing the file.

Write Text to a New File

What do you do if you want only part of the file in the editing buffer placed in a UNIX system file?

Many of the commands in **ex** will accept a line number or a range of line numbers typed in before the command **w**. Try to write the third line of the buffer to a file named *three*.

Type in: **:3w three<CR>**

Notice the system response.

"three" [New file] 1 line, 20 characters

The "." is the special character that indicates the number of the current line.

Type in: **:.w junk<CR>**

A new file called *junk* will be created containing only the current line in the **vi** buffer.

You can also specify the range of lines. To write lines 23 through 37 to a file, type in:

23,37w newfile<CR>

Finding the Line Number

If you want to specify a range of lines, you can find out the line number of that line by moving the cursor to that line.

Type in: **:.=<CR>**

The editor will come back with the response that is the number of that line.

```
If you want to know the number
of this line, type in :.=<CR>

:.=
```

As soon as you press RETURN, the bottom line will clear and give you the number of the line in the buffer.

```
If you want to know the number
of this line, type in :.=<CR>

34
```

You can move the cursor to any line in the buffer by typing in a ":" and the line number.

:n<CR> Go to the "nth" line of the buffer.

Deleting the Rest of the Buffer

One of the easiest ways to delete all the lines from the current line to the end of the buffer is to use the line editor command to delete lines.

Type in: **:.,$d<CR>**

The "." is the current line, and the last line is **$**.

Adding a File to the Buffer

If you have a file with some data or text in it that you would like to add below a specific line in the editing buffer, you can do so with the **:r** command. To read in the file *data* place the cursor on the line above the desired insertion.

Type in: **:r data<CR>**

You may also specify the line number instead of moving the cursor. Insert the file *data* below line 56 of the buffer.

Type in: **:56r data<CR>**

Do not be afraid to experiment, **<u>** will undo the **ex** commands too.

Making Global Changes

One of the most powerful commands in **ex** is the global command. The global command is given here to help those users who are familiar with the line editor. Even if you are not familiar with a line editor, you may want to try the command on a test file.

If you had typed in several pages of text about the DNA molecule, calling its structure a "helix", you would have to change each occurrence of the word "helix" to "double helix". This could be a long

involved process searching for each one and probably using the "." command of **vi** to repeat the change. If you are sure you want every "helix" changed, you can use the global command of **ex**. You need to understand a series of commands to do this. Let's take one at a time.

:g/characters<CR>

Search for these exact characters.

Type in: **:g/helix<CR>**

The line editor does a global search for the first instance of the characters "helix" on a line.

:s/text/new words/<CR>

This is the substitute command. Instead of writing over the word **text**, as the screen editor would have done, the line editor searches for the first instance of the characters **text** on the current line, and changes them to **new words**. You must tell **ex** what word you are looking for and it must appear between the first two delimiters, /. It will then replace only those exact characters with the exact characters, **new words**, between the last two delimiters.

:s/text/new words/g<CR>

By adding a "g" at the end of the last delimiter of this command line, **ex** will change every occurrence on the current line.

:g/helix/s//double helix/g<CR>

This command line searches for the word **helix**. Each time **helix** is found, the substitute command substitutes **double helix** for every instance of **helix** on that line. The delimiters after the **s** do not need to have **helix** typed in again. The command remembers the word from the delimiters after the global command **g**.

This is a very powerful command. If it is confusing to you, but you still want to add it to your **vi** command knowledge, read *Chapter 5* on the line editor **ed** for a more detailed explanation of the global and substitution commands.

SUMMARY OF LINE EDITOR COMMANDS

:	Indicates that the next commands are line editor commands.
:sh<CR>	Temporarily return to the shell to perform some shell commands.
<^d>	Escape the temporary shell and return to edit the current window of **vi**.
:n<CR>	Go to the "nth" line of the buffer.
:x,z**w data<CR>**	Write lines from the number "x" through the number "z" into a new file called *data*.
:$<CR>	Go to the last line of the buffer.
:.,$d<CR>	Delete all the lines in the buffer from the current line to the last line.
:r shell.file<CR>	Insert the contents of *shell.file* under the current line of the buffer.
:s/text/new words/<CR>	Replace the first instance of the characters **text** on the current line with **new words**.
:s/text/new words/g<CR>	Replace every occurrence of **text** on the current line with **new word**.
:g/text/s//new word/g<CR>	Change every occurrence of **text** to **new word**.

QUITTING VI

There are six basic command sequences to quit the **vi** editor.

<ZZ> Write the contents of the **vi** buffer to the UNIX system file currently being edited and quit **vi**.

:wq<CR> Write the contents of the **vi** buffer to the UNIX system file currently being edited and quit **vi**.

:w filename<CR>
:q<CR> Write the temporary buffer to a new file named *filename* and quit **vi**.

:w! filename<CR>
:q<CR> Overwrite an existing file called *filename* with the contents of the buffer and quit **vi**.

:q!<CR> Quit **vi** without writing to the shell file.

:q<CR> Quit **vi** without writing the buffer to a UNIX system file. This command, without the write command **w**, can only be used in special cases, such as the **view** command discussed in the next section, or if the buffer has not been changed.

The commands that are preceded by a ":" are line editor commands.

The **<ZZ>** command and **:wq** command sequence both write the buffer to a UNIX system file, then quit **vi**, and return you to the shell command level. You have tried the **<ZZ>** command, now try to exit **vi** with **:wq**.

Type in: **:wq<CR>**

The system response is the same as it is for the **<ZZ>** command. It gives you the name of the file, and the number of lines and characters in the file.

vi remembers the file name of the file currently being edited, so you do not have to reiterate the file name when you want to write the buffer of the editor back into that file. What do you do if you want to give the file a different name?

If you want to write to a file called *junk*:

Type in: **:w junk<CR>**

After you write to a new file, you can leave **vi** by just typing in the **:q**.

Type in: **:q<CR>**

If you try to write to a file called *letter* that already exists in the shell, you will receive a warning:

"letter" File exists - use "w! letter" to overwrite

Type in: **:w! letter<CR>**

You will erase the current file called *letter* and overwrite it with the new file.

If you began editing a file called *memo*, made some changes to the file, and then decided you didn't want to make the changes, or you accidentally pressed a key that gave **vi** a command you could not undo, you can leave **vi** without writing to the file.

Type in: **:q!<CR>**

SUMMARY OF QUIT COMMANDS

<ZZ>	Write the file and quit **vi**.
:wq<CR>	Write the file and quit **vi**.
:w filename<CR> **:q<CR>**	Write the editing buffer to a new file named *filename* and quits **vi**.

(Continued on next page)

SUMMARY OF QUIT COMMANDS *(continued)*

:w! filename<CR>	
:q<CR>	Overwrite an existing file called *filename* with the contents of the editing buffer and quits **vi**.
:q!<CR>	Quit **vi** without writing to the buffer.
:q<CR>	Quit **vi** without writing the buffer to a UNIX system file.

SPECIAL OPTIONS FOR vi

The **vi** command has some special options. It allows you to:

- Recover a file lost by an interrupt to the UNIX system,
- Place several files in the editing buffer and edit each in sequence, and
- View the file with the **vi** cursor positioning commands.

Recovering a File Lost by an Interrupt

There are times when an interrupt or a disconnect will cause the system to exit the **vi** command without writing the temporary buffer to the UNIX system file. Or, you may become confused or have a problem with the **vi** editor that you cannot solve. If that happens, one solution is simply to hang up, or disconnect from the UNIX system. In both of these cases, the UNIX system will store a copy of the buffer for you. When you log back into the UNIX system you will want to restore the file with the **−r** option for the **vi** command:

Type in: **vi −r filename<CR>**

The changes you made to the file *filename*, before the interrupt occurred, are now in the **vi** buffer. You can continue editing the file, or you can write the file and quit **vi**. The **vi** editor will remember the file name and write to that file.

Editing Multiple Files

If you wish to edit more than one file in the same editing session, type in the **vi** command followed by each file name.

Type in: **vi file1 file2<CR>**

vi will respond by telling you how many files you are going to edit.

2 files to edit

After you have edited the first file, *file1*, you need to write the changes to the shell file.

Type in: **:w<CR>**

The system response to the **:w <CR>** command will be a message at the bottom of the screen giving the name of the file, and how many lines and characters are in that edited file. Then you must ask for the next file in the editing buffer with the **:n** command.

Type in: **:n<CR>**

The system response to the command **:n<CR>** is a notice at the bottom of the screen with the name of the next file to be edited and the character and line count of that file.

Pick two of the files in your current directory and enter **vi** to place the two files in the editing buffer at the same time. Notice the system responses to the commands at the bottom of the screen.

SUMMARY OF SPECIAL OPTIONS FOR vi

vi file1 file2 file3<CR>	Enter three files into the **vi** buffer to be edited. Those files are *file1, file2,* and *file3.*
:w<CR> **:n<CR>**	Write the current file and call the next file in the buffer.
vi −r file1<CR>	Restore the changes made to the file *file1.*

EXERCISE 6

6-1. Try to restore a file lost by an interrupt.

Enter **vi**, create some text in a file called *exer6.*

Turn off your terminal without writing to a file or leaving **vi**.

Log back in to your terminal.

Try to get back into **vi** and edit the *exer6* file.

6-2. Place *exer1* and *exer2* in the **vi** buffer to be edited.

Write *exer1* and call in the next file in the buffer, *exer2.*

Write *exer2* to a file called *junk.*

Quit **vi**.

6-3. Try out the command:

vi exer*<CR>

What happens? To quit **vi**:

Type in: **ZZ ZZ**

6-4. Look at *exer4* in read only mode.

Scroll forward.

Scroll down.

Scroll backward.

Scroll up.

Quit and return to the shell.

CHANGING YOUR ENVIRONMENT

If you are going to edit with **vi** you will want to change your login environment so that you do not have to reconfigure your terminal each time you login. Your login environment is controlled by a file in your login directory called the *.profile*. The *.profile* is explained in more detail in the shell tutorial in *Chapter 7*.

You are about to edit your *.profile* that sets up your environment each time you login. If you are concerned that you might cause a problem with your *.profile* in the editing process, you may want to keep a backup copy of your original *.profile* for safekeeping.

From your login directory, type in:

cp .profile safe.profile<CR>

Now that you have a copy of your *.profile* in a safe place, *safe.profile*, you can edit your *.profile* just like any other file in **vi**.

Type in: **vi .profile<CR>**

Go to the last line of the file, ignoring all the lines currently in the file.

Type in: **G**

You are going to add two lines to the bottom of the file, the same terminal configuration you typed in at the beginning of your login session so that you could enter **vi**.

Type in: **<o>**

Now you are ready to append text to the end of the file.

Type in: **TERM=code<CR>**
export TERM<CR>

Remember "code" is the special code characters for your type of terminal.

Write and quit **vi**. Now, the next time that you log into the UNIX system **TERM** is automatically set and you can immediately begin editing with **vi**.

Setting the Automatic Carriage Return

If you want an automatic carriage return, create a new file *.exrc*. The *.exrc* file controls the editing environment for **vi**. There are several options you can set in this file. If you want to know more about *.exrc*, read the *Editing Guide*. (See *Appendix A*.)

Type in: **vi .exrc<CR>**

Add one line to this file.

Type in: **wm=**n<CR>

"n" is the number of characters from the right side of the screen where the carriage return will occur. If you want a carriage return at 20 characters from the right edge of the screen,

Type in: **wm=20<CR>**

Write and quit that file. The next time you login this file will give you an automatic carriage return.

You can check on these settings, the terminal setting and the wrapmargin (automatic carriage return) when you are in **vi**.

Type in: **:set<CR>**

vi will tell you the terminal type and the wrapmargin. You can also use the **:set** command to create or change the wrapmargin. Try experimenting with it.

Now you know the basics of **vi**! Experiment with the commands, find the ones that work best for you.

ANSWERS TO EXERCISES

There is often more than one way to perform a task in **vi**. If the way you tried worked, then your answer is correct. Below are suggestions for performing the task given in the exercise.

Exercise 1

1-1. Look up your terminal code with the following command. Type in:

grep "your type of terminal" /etc/termcap<CR>

The first two letters of of the system response are your terminal code. Type in:

TERM=code<CR>
export TERM<CR>

1-2. Type in:

vi exer1<CR>
<a>
This is an exercise!**<CR>**
Up, down**<CR>**
left, right,**<CR>**
build your terminal's**<CR>**
muscles bit by bit.**<ESC>**

1-3. Use the **<k>** and the **<h>** commands.

1-4. Use **<x>**.

1-5. Use the **<j>** and **<l>** commands.

1-6. Type in:

<a> <CR>
and byte by byte**<ESC>**

Use **<j>** and **<l>** to move to the last line and character of the file. Use **<a>** to add text. **<CR>** will create the new line. **<ESC>** will end the create mode.

1-7. Type in:

ZZ

1-8. Type in:

vi exer1<CR>

System response:
"exer1" 6 lines, 100 characters

Exercise 2

2-1. Type in:

vi exer2<CR>
<a>1<CR>
2<CR>
3<CR>
.
.
.
48<CR>
49<CR>
50<ESC>

2-2. Type in:

<^f>
<^b>
<^u>
<^d>

Notice the line numbers as the screen changes.

2-3. Type in:

<G>
<o>
123456789 123456789<ESC>

2-4. $ = end of line
0 = first character in the line

2-5. Type in:

<^>
<w>
<b>
<e>

2-6. Type in:

<1G>
<M>
<L>
<H>

2-7. Type in:

/8
<n>
/48

Exercise 3

3-1. Type in:

vi exer3<CR>

3-2. Type in:

<a> Append text **<CR>**
Insert text**<CR>**
a computer's **<CR>**
job is boring.**<ESC>**

3-3. Type in:

<O>
financial statement and**<ESC>**

3-4. Type in:

<3G>
<i>Delete text**<CR> <ESC>**

The text in your file now reads:

Append text
Insert text
Delete text
a computer's
financial statement and
job is boring.

3-5. The current line is "a computer's". To create a line of text below that line use the **<o>** command.

3-6. The current line is "byte of the budget".
<G> will put you on the bottom line.
<A> will begin appending at the end of the line.
<CR> will create the new line.
Then, type in the text "But, it is an exciting machine."
<ESC> ends the append mode.

3-7. Type in:

<1G>
/text
<i>some**<space bar> <ESC>**

3-9. **<ZZ>** will write the buffer to *exer3* and put you in the command mode of the shell.

Exercise 4

4-1. Type in:

vi exer4<CR>
<a> When in the course of human events**<CR>**
there are many repetitive, boring**<CR>**
chores, then one ought to get a**<CR>**
robot to perform those chores.**<ESC>**

4-2. Type in:

<2G>
<A> tedious and unsavory**<CR>**
<8BS>
<ESC>

Press **<h>** until you get to the "b" of "boring" then press **<dw>**. Or, you could have used **<6x>**.

4-3. You are at the second line. Type in:

<2j>
<I> congenial and computerized**<ESC>**
<dd>

To delete the line and leave it blank, type in:

<0> (zero to place you at the beginning of the line)
<D>

<H>
<3dd>

4-4. Write and quit **vi**.

<ZZ>

Remove the file.

rm exer4<CR>

Exercise 5

5-1. Type in:

vi exer2<CR>
<8G>
<cc> END OF FILE <ESC>

5-2. Type in:

<1G>
<8"zyy>
<G>
<"zp>

5-3. Type in:

<8G>
<cc> 8 is great**<ESC>**

5-4. Type in:

<18G>
<.>

5-5. Type in:

</FI>
<cw> EXERCISE**<ESC>**

<?OF>
<R>TO<ESC>

Exercise 6

6-1. Type in:

vi exer6<CR>
<a> (append several lines of text)
<ESC>

Turn off the terminal.

Turn on the terminal.
Log into the UNIX system. Type in:

vi −r exer6<CR>
:wq<CR>

6-2. Type in:

vi exer1 exer2<CR>
:w<CR>
:n<CR>

:w junk<CR>
ZZ

6-3. Type in:

vi exer*<CR>

(Response)
8 files to edit (**vi** calls in all files with names that begin with *exer*.)

ZZ
ZZ

6-4. Type in:

view exer4<CR>
<^f>
<^d>
<^b>
<^u>

Chapter 7

SHELL TUTORIAL

PAGE

MAKING LIFE EASIER IN THE SHELL 7-1

HOW TO READ THIS TUTORIAL 7-2

SHELL COMMAND LANGUAGE 7-3
- Special Characters in the Shell 7-3
  - Metacharacters 7-3
  - Metacharacter that Matches All Characters 7-4
  - Metacharacter that Matches One Character 7-6
  - Metacharacters that Match One of a Specific Range of Characters 7-8
  - Commands in the Background Mode 7-9
  - Sequential Execution 7-10
  - Turning Off Special Character Meaning 7-11
  - Turning Off Special Characters by Quoting 7-11
  - Redirecting Input and Output 7-14
  - Redirecting Input 7-14
  - Redirecting Output 7-15
  - Pipes 7-19
  - Command Output Substitution 7-22
- Executing and Terminating Process 7-23
  - Running Commands at a Later Time 7-23
  - Obtaining the Status of Running Processes 7-28
  - Terminating Active Processes 7-29
  - Using the No Hang Up Command 7-30

COMMAND LANGUAGE EXERCISES 7-31

SHELL PROGRAMMING 7-32
- Getting Started 7-32
  - Creating a Simple Shell Program 7-34
  - Executing a Shell Program 7-35

PAGE

Creating a bin Directory for Executable Files 7-36
Variables 7-38
Positional Parameters 7-39
Parameters with Special Meaning 7-43
Variable Names 7-46
Assign Values to Variables 7-48
Assign Variable by the Read Command 7-48
Substitute Command Output for the Value of a Variable 7-52
Assign Values with Positional Parameters 7-54
Shell Programming Constructs 7-55
Comments 7-56
The Here Document 7-56
Using ed in a Shell Program 7-58
Looping 7-60
The for Loop 7-61
The while Loop 7-64
Conditional Constructs if...then 7-66
The Shell Garbage Can /dev/null 7-67
The test Command for Loops 7-69
The Conditional Construct case...esac 7-72
Unconditional Control Statement break 7-75
Debugging Programs 7-77
Modifying Your Login Environment 7-80
What is a .profile? 7-80
Adding Commands to .profile 7-81
Setting Terminal Options 7-82
Using Shell Variables 7-83
Conclusion 7-85

SHELL PROGRAMMING EXERCISES 7-85

ANSWERS TO EXERCISES 7-88
Command Language Exercises 7-88
Shell Programming Exercises 7-89

Chapter 7

SHELL TUTORIAL

MAKING LIFE EASIER IN THE SHELL

You have used the shell to interact with the UNIX system by typing in commands that give you information, such as **who**, or commands that perform a task, such as **sort**. This chapter introduces some methods and commands that will help expedite the day-to-day tasks that you perform in the shell.

The first part of the tutorial, *Shell Command Language*, introduces some basic shortcuts and commands to help you perform tasks in the UNIX system quickly and easily. The second part of the tutorial, *Shell Programming*, shows you how to put these tasks into a file and call on the shell to execute the commands in the file while you go get a cup of coffee. The following basics are covered:

- How to use some special characters in the shell,
- How to redirect input and output,
- How to execute and terminate processes,
- How to create and execute a simple shell program,
- How to use variables in a shell program,
- How to use shell programming constructs for looping, conditional execution, and unconditional execution,
- How to locate problems and debug a shell program, and
- How to modify your login environment by editing the file called *.profile*.

If, after you have read this tutorial, you want to learn more advanced concepts in shell programming, read *UNIX System Shell Commands and Programming*. (See *Appendix A*.)

HOW TO READ THIS TUTORIAL

Log into your UNIX system and try the examples as you read the text. Experiment with the concepts and perhaps combine them into a shell program. Often, there is more then one correct way to write a shell program. You may discover a different method. If your shell program works, if it performs the task, then it is a correct method.

Here is a quick review of the text conventions mentioned in *Chapter 2* that are used throughout this book.

bold command	(Type in the command line exactly as shown.)
italic response	(The system's response to a command.)
< >	(Commands that are typed in, but not displayed on your terminal, are enclosed in < >.)
^g	(A control character, hold down the control key CTRL while your press "g".)

A display screen like the one above is used to illustrate the commands and the text of the shell programs. You may not be working on a terminal with a screen. This will not affect the shell tasks that you perform or shell programs that you create. The lines that you type in and the system responses should be the same.

SHELL COMMAND LANGUAGE

Special Characters in the Shell

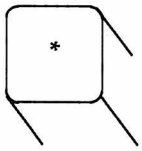
The shell language has special characters that give you some shortcuts for performing tasks in the shell. These special characters are listed below and are discussed in this section of the tutorial.

* ? [ ]	These are metacharacters. A metacharacter is a character that has a special meaning in shell command language. These metacharacters give you shortcuts for file names.
&	This character places commands in the background mode. While the shell is performing the commands in the background, your terminal is free for you to work on other tasks.
;	This character allows you to type in several commands on one line. Each command must be followed by a ; . When you type in the <CR>, each command will execute sequentially from the beginning of the line to the end of the line.
\	This character allows you to turn off the meaning of special characters such as *, ?, [ ], & and ; .
" ... " ' ... '	Both double and single quotes turn off the delimiting meaning of the space, and the special meaning of special characters. However, double quotes will allow the characters $ and \ to retain their special meaning. (The $ and \ are discussed later in this chapter and are important for shell programs.

Metacharacters

The meaning of the metacharacters is similar to saying "etc. etc. etc.", "all of the above", or "one of these". Using metacharacters for all or part of a file name is called file name generation. It is a quick and easy way to refer to file names.

Metacharacter That Matches All Characters



This metacharacter matches "all", any string of characters, including no characters at all.

The * alone refers to all the file names in the current directory, the directory you are in now. To see the effect of the *, try the next command.

Type in: **echo *<CR>**

The **echo** command displays its arguments on your terminal. The system response to **echo *** should have been a listing of all file names in the current directory. However, unlike **ls**, the file names were displayed in horizontal lines instead of a vertical listing.

Since you may not have used the **echo** command before, here is a brief recap of the command.

Command Recap

echo - write any arguments to the output

command	*options*	*arguments*
echo	none	**any character**

Description: **echo** writes arguments, which are separated by blanks and ended with **<CR>**, to the output.

Remarks: In shell programming, **echo** will be used to issue instructions, to redirect words or data into a file, and to pipe data into a command. All of these uses will be discussed later in this chapter.

Problem:
Be very careful with * because it is a powerful character. If you type in **rm** * you will erase all the files in your current directory.

The * metacharacter is also used to expand file names in the current directory. If you have written several reports and have named them:

report
report1
report1a
report1b.01
report25
report316

then

report*

refers to all six reports in the current directory. If you want to find out how many reports you have written, you could use the **ls** command to list all the reports that begin with the letters **report**.

```
$ ls report*<CR>
report
report1
report1a
report1b.01
report25
report316
$
```

The * refers to any characters after the letters **report**, including no letters at all. Notice that * calls the files in numerical and alphabetical order. A quick and easy way to print out all of those reports in order is:

Type in: **pr report*<CR>**

Choose a character that your file names have in common, such as an **a**, and list all those files in the current directory.

Type in: **ls *a*<CR>**

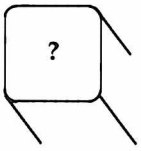
The * can be placed anywhere in the file name.

Type in: **ls F*E<CR>**

This command line would list all of the following files in order:

F123E
FATE
FE
Fig3.4E

Metacharacter That Matches One Character



This metacharacter matches any single character.

The **?** metacharacter replaces any one character of a file name. If you have created text for several chapters of a book, but you only want to list the chapters you have written through **chapter9**, you would use the **?** .

```
$ ls chapter?<CR>
chapter1
chapter2
chapter5
chapter9
$
```

Although **?** matches any one character, you can use it more than once in a file name. To list the rest of the chapters up through **chapter99**, type in:

ls chapter??<CR>

Of course, if you want to list all the chapters in the current directory you would use **chapter***.

Problem:
Sometimes when you **mv** or **cp** a file you accidentally press a character that does not print out on your terminal as part of the file name when you do an **ls**. If you try to **cat** that file, you get an error message. The * and ? are very useful in calling up the file and moving it to the correct name. Try the following example.

1. Make a very short file called *trial*.

2. Type in: **mv trial trial<^g>1<CR>**

   Remember to type in **<^g>** you hold down the CTRL key and press the "g" key.

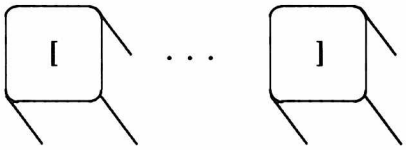
3. Type in: **ls trial1<CR>**

```
$ ls trial1<CR>
trial1 not found
$
```

4. Type in: **ls trial?1<CR>**

```
$ ls trial?1<CR>
trial1
$ mv trial?1 trial1<CR>
$ ls trial1<CR>
trial1
$
```

Metacharacters That Match One of a Specific Range of Characters



The shell matches one of the specified characters or range of characters within the brackets.

Characters enclosed in **[ ]** act as a specialized form of the **?** . The shell will match only one of the characters enclosed in the brackets in the position specified in the file name. If you use **[crf]** as part of a file name, the shell will look for **c**, or **r**, or **f**.

```
$ ls [crf]at<CR>
cat
fat
rat
$
```

The shell will also look for a range of characters within the brackets. For **chapter[0-5]** the shell looks for the files named *chapter0* through *chapter5*. This is an easy way to print out only certain chapters at one time.

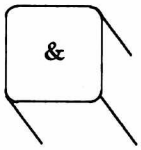
Type in: **pr chapter[2-4]<CR>**

This command will print out the contents of *chapter2, chapter3,* and *chapter4* in that order.

The shell will also look for a range of letters. For **[A-Z]**, the shell will look for uppercase letters, or for **[a-z]**, the shell will look for lowercase letters.

Try out each of these metacharacters on the files in your current directory.

Commands in the Background Mode



This character, placed at the end of a command line, runs a task in background mode.

Some shell commands take considerable time to execute. It is convenient to let these commands run in background mode to free your terminal so that you can continue to type in other shell tasks. The general format for a command to run in background mode is:

command &<CR>

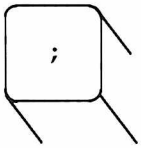
The **grep** command can perform long searches that may take a lot of time. If you place the **grep** command in a background mode, you can continue doing some other task at your terminal while the search is being done by the shell. In the example below, the background mode is used while all the files in the directory are being searched for the characters **word**. The **&** is the last character after the command.

```
$ grep word * &<CR>
21940
$
```

21940 is the process number. This number is essential if you want to stop the execution of a background command. This will be discussed in *Executing and Terminating Processes.*

In the next section of this tutorial you will see how to redirect the system response of the **grep** command into a file so that it does not display on your terminal and interrupt your current work. Then, you can look at the file when you have finished your task.

Sequential Execution



The shell performs sequential execution of commands typed on one line and separated by a ; .

If you want to type in several commands on one line, you must separate each command with a ; . The general format to place **command1**, **command2**, and **command3** on one command line is the following:

command1; command2; command3<CR>

Sequential execution is very useful if you need to execute several shell commands while you are in the line editor **ed**. (See the section on *Other Useful Commands and Information* in *Chapter 5*.) Try out the ; . Type in several commands separated by a ; . Notice that, after you press **<CR>**, the system responds to each command in the order that they appear on the command line.

Type in: **cd; pwd; ls; ed trial<CR>**

The shell will execute these commands sequentially:

1. **cd** Change to login directory.

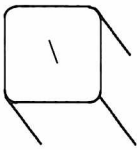
2. **pwd** Print the path of the current directory.

3. **ls** List the files in the current directory.

4. **ed trial** Enter the line editor **ed** and begin editing the file *trial*.

Did you notice the rapid fire response to each of the commands? You may not want these responses to display on your terminal. The section on *Redirecting Output* will show you how to solve this problem.

Turning Off Special Character Meaning

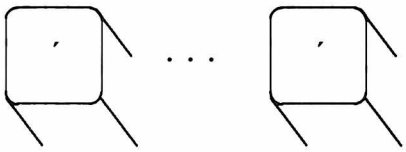


The backslash turns off the special meaning of a metacharacter.

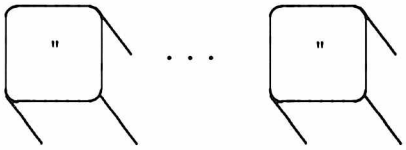
How do you search for one of the special characters in a file? Type in a backslash just before you type in the metacharacter. The backslash turns off the special meaning of the next character that you type in. Create a file called *trial* that has one line containing the sentence "The all * game". Search for the * character in the file *trial*.

```
$ grep \* trial<CR>
The all * game
$
```

Turning Off Special Characters by Quoting



All special characters enclosed in single quotes lose their special meaning.



All special characters except $, \, and ` lose their special meaning when they are in double quotes.

The special characters in the shell lose their special meaning when they are enclosed by quotes. The single quote turns off the special meaning of any character. The double quote will turn off the special meaning of any character except **$** and `. The **$** and ` are very important characters in shell programming.

A delimiter separates arguments, telling the shell where one argument ends and a new one starts. The space has a special meaning to the shell because it is used as a delimiter between arguments of a command.

The **banner** command uses spaces to delimit arguments. If you have not used the **banner** command, try it out. The system response is rather surprising.

Type in: **banner happy birthday to you<CR>**

Was each word displayed in large poster sized letters?

Now put quotes around **to you**.

Type in: **banner happy birthday "to you"<CR>**

Notice that **to** and **you** appear on the same poster display line. The space between the **to** and the **you** has lost its special meaning as a delimiter.

Since you may not have used the **banner** command before, the following is a quick recap of that command. You may find that you do not have access to the **banner** command. Not all systems have all the commands referenced in this chapter. If you cannot access a command, check with your system administrator.

Command Recap

banner - make posters

command	*options*	*arguments*
banner	none	**characters**

Description: Displays arguments, up to ten characters on a poster-sized line, in large letters.

Remarks: Later in this chapter you will learn how to redirect the **banner** command into a file to be used as a poster.

If you use single quotes in the argument for the **grep** command, the space loses the meaning of a delimiter. You can search for two words. The line, **The all * game** is in your file *trial*. Look for the two words **The all** in the file *trial*.

```
$ grep 'The all' trial<CR>
The all * game
$
```

Try turning off the special character meaning of the * using single quotes.

```
grep '*' trial<CR>
The all * game
$
```

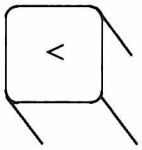
If you want to know more about quoting, read the *UNIX System User Reference Manual* pages on the **sh** command.

Redirecting Input and Output

The redirection of input and output are important tools for performing many shell tasks and programs.

Redirecting Input

You can redirect the text of a file to be the input for a command.



This character redirects the contents of a file into a command.

The general format to redirect the contents of a file into a command is shown below.

command < filename<CR>

If you write a report to your boss, you probably do not want to type in the **mail** command and then type in your text. You want to be able to put your report in an editor and correct errors. You want to run the file through the **spell** command to make sure there are no misspelled words. You can **mail** a file containing your report to another login using the redirection symbol. In the example below, a file called *report* is checked for misspelled words and then redirected to be the input to the **mail** command and mailed to login **boss**.

```
$ spell report<CR>
$
$ mail boss < report<CR>
$
```

Since the only response to the **spell** command is the prompt, there are no misspelled words in *report*. The **spell** command is a useful tool that gives you a list of words that are not in a dictionary spelling list. The following is a brief recap of **spell**.

Command Recap

spell - find spelling errors

command	*options*	*arguments*
spell	available*	**filename**

Description: **spell** collects words from the specified file or files and looks them up in a spelling list. Words that are not on the spelling list are displayed on your terminal.

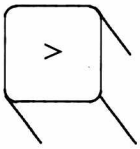
Options: **spell** has several options, including one for checking the British spelling.

Remarks: The misspelled words can be redirected into a file. See the redirection symbol > discussed next.

* See the *UNIX System User Reference Manual* for all available options and an explanation of their capabilities.

Redirecting Output

You can redirect the output of a command to be the contents of a file. When you redirect output into a file, you can either create a new file, append the output to the bottom of a file, or you can erase the contents of an old file and replace it with the redirection output.



This character redirects the output of a command into a file.

The single redirection symbol > will create a new file, or it will erase an old file and replace the contents with new output. The general format to redirect output is shown below.

command > filename<CR>

If you want the **spell** command list of misspelled words placed in a file instead of displayed on your terminal, redirect **spell** into a file. In the example, **spell** searches the file *memo* for misspelled words and places those words in the file *misspell.*

```
$ spell memo > misspell<CR>
$
```

The **sort** command can be redirected into a file. Suppose a file called *list* contains a list of names. In the next example, the output of the **sort** command lists the names alphabetically and redirects the list to a new file *names.*

```
$ sort list > names<CR>
$
```

Problem:
Be careful to choose a new name for the file that will contain the alphabetized list. The shell first cleans out the contents of the file

that is going to accept the redirected output, then it sorts the file and places the output in the clean file. If you type in

sort list > list<CR>

the shell will erase *list* and then sort nothing into *list*.

Problem:
If you redirect a command into a file that exists, the shell will erase the existing file and put the output of the command into that file. No warning is given that you are erasing an existing file. If you want to assure yourself that there is not an existing file, first execute the **ls** command with the file name as an argument.

If the file exists, **ls** will list the file. If the file does not exist, **ls** will tell you the file was not found in the current directory.

```
$ ls filename<CR>
filename
$ ls junk<CR>
junk not found
$
```

The double redirection symbol >> appends the output of a command after the last line of a file.

The general format to append output to a file is:

command >> filename<CR>

In the next example, the contents of *trial2* are added after the last line of *trial1* by redirecting the **cat** command output of *trial2* into *trial1*.

The first command, **cat trial1**, displays the contents of *trial1*. Then, **cat trial2** displays the contents of *trial2*. The third command line, **cat trial2 >> trial1**, adds the contents of *trial2* to the bottom of file *trial1*, and **cat trial1** displays the new contents of *trial1*.

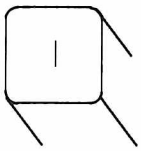
```
$ cat trial1<CR>
hello
this is a trial
This is the last line of this file
$
$ cat trial2<CR>
Add this to file trial1
This is the last line of file trial2
$
$ cat trial2 >> trial1<CR>
$ cat trial1<CR>
hello
this is a trial
This is the last line of this file
Add this to file trial1
This is the last line of file trial2
$
```

In the section on *Special Characters*, one of the examples showed how to execute the **grep** command in background mode with **&**. Now, you can redirect the output of that command into a file called *wordfile*, and then look at the file when you have finished your current task. The **&** is the last character of the command line.

```
$ grep word * > wordfile &<CR>
$
```

Pipes

The | character is called a pipe. It redirects the output of one command to be the input of another command.



This character directs the output from one command to be the input of the next command.

If two or more commands are connected by a pipe, |, the output of the first command is "piped" into the next command as the input for that command.

The general format for the pipe line is:

command1 | command2 | command3<CR>

The output of **command1** is used as the input of **command2**. The output of **command2** is then used as the input for **command3**.

You have already tried the banner display on your terminal. The pipe can be used to send a banner birthday greeting to someone by electronic mail.

If the person using login **david** has a birthday, pipe the banner display of happy birthday into the **mail** command.

Type in: **banner happy birthday | mail david<CR>**

Login **david** will get a banner display in his electronic mail.

The **date** command gives you the date and the time. Since you may not have used the **date** command before, a brief recap of **date** follows.

Command Recap

date - display the date and time

command	*options*	*arguments*
date	**+%m%d%y*** **+%H%M%S**	available*

Description: **date** displays the current date and time on your terminal.

Options: +% followed by m for month, d for day, y for year, H for hour, M for month, and S for second will echo these back to your terminal. You can add an explanation to these such as:

date '+%H:M is the time'

Remarks: If you are working on a small computer system in which you are acting as both user and system administrator, you may be able to set the date and time using optional arguments to the **date** command. Check your reference manual for details. When working in a multiuser environment, the arguments are available only to the system administrator.

* See the *UNIX System User Reference Manual* for all available options and an explanation of their capabilities.

Try out the **date** command on your terminal.

```
$ date<CR>
Mon Nov 25 17:57:21 CST 1985
$
```

Notice that the time is given from the 12th character through the 19th character. If you want to know just the time and not the date, you can pipe the output of the **date** command into the **cut** command. The **cut** command looks for characters only in a specified part of each line of a file. If you use the **−c** option, **cut** will choose only those characters in the specified character positions. Character positions are counted from the left. To display only the time on your terminal pipe the output of the **date** command into the **cut** command asking for characters 12 through 19.

```
$ date | cut −c12-19<CR>
18:08:23
$
```

Several pipes can be used in one command line. The output of the example can be piped into the **pr** command.

Type in: **date | cut −c12-19 | pr<CR>**

Try each of these examples. Check the system response.

Later in this chapter, you will write a shell program that will give you the time.

Since you may not have used the **cut** command until now, a brief recap of that command follows next.

Command Recap

cut - cut out selected fields of each line of a file

command	*options*	*arguments*
cut	−clist −flist [−d]	**file1 file2**

Description: **cut** will cut out columns from a table or fields from each line of a file.

Options: −**c** lists the number of character positions from the left. A range of numbers such as characters 1-9 can be specified by −**c1-9**

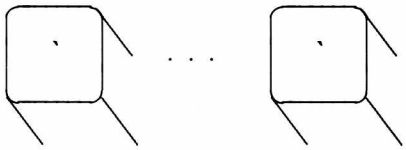
−**f** lists the number of fields from the left separated by a delimiter described by −**d**.

−**d** gives the field delimiter for −**f**. The default is a tab. If the delimiter is a colon, this would be specified by −**d :** .

Remarks: If you find the **cut** command useful, you may also want to use the **paste** command and the **split** command.

Command Output Substitution

The output of any command line or shell program that is enclosed in back quotes, `, can be substituted anywhere on a shell command line. In the section on *Shell Programming*, you will substitute the output of a command line as the value for a variable.



Substitute the output of the command line in back quotes.

The output of the time command can be substituted for the argument in a banner printout.

Type in: **banner `date | cut −c12-19`<CR>**

Did you get a banner display of the time?

Executing and Terminating Processes

Running Commands at a Later Time

When you type in a command line at your terminal, the UNIX system tries to execute that command immediately. It is possible to tell the system to execute those commands at another time with the **batch** or the **at** command. End the commands with **<^d>** to let the shell know you have finished listing the commands to be executed.

The **batch** command is useful if you are running a process or shell program that uses a longer then normal amount of system time. The **batch** command submits a "batch" job, which consists of the commands to be executed, to the system. The job is put in a queue, and then the job is run when the load on the system falls to an acceptable level. This frees the system to rapidly respond to other input by yourself or others on the system.

The general format for **batch** is:

```
batch<CR>
first command<CR>
      .
      .
      .
last command<CR>
<^d>
```

If there is only one command line, it may follow the **batch** command.

```
batch command line<CR>
<^d>
```

The next example uses the **batch** command to execute the **grep** command at a convenient time. When the system can execute that command and still respond quickly to other users, it will execute the **grep** command to search all the files for the word **dollar**, and redirect the output into the file *dol–file*. Using the **batch** command is a courtesy to other users sharing your UNIX system.

```
$ batch grep dollar * > dol-file<CR>
<^d>
job 155223141.b at Mon Dec 7 11:14:54 1983
$
```

A brief recap of the **batch** command follows.

Command Recap

batch - execute commands at a later time

command	*options*	*arguments*
batch	none	**command lines**

Description: **batch** submits a "batch job", which is placed into a queue and executed when the load on the system falls to an acceptable level.

Remarks: The list of commands must end with a **<^d>** to tell the system the last command has been typed in for this batch job.

The **at** command gives the system a specific time that the commands are to be executed. The general format for the **at** command is:

```
at time<CR>
first command<CR>
      .
      .
      .
last command<CR>
<^d>
```

The **time** must first give the time of day and then the date, if the date is not today.

If you are afraid you will forget login *david*'s birthday, you can use the **at** command to make sure the banner birthday greeting will arrive on his birthday.

```
$ at 8:15am Feb 27<CR>
banner happy birthday | mail david<CR>
<^d>
job 453400603.a at Mon Feb 27 08:15:00 1984
$
```

Both the **batch** and **at** commands give you a job number. If you decide you do not want to execute the commands currently waiting in a **batch** or **at** job queue, you can erase those jobs with the **−r** option of the **at** command and the job number. The general format is:

at −r jobnumber<CR>

Try erasing the previous **at** job for the happy birthday banner.

Type in: **at −r 453400603.a<CR>**

If you have forgotten the job number, the **at −l** command will give you the current jobs in the **batch** or **at** queue.

```
$ at -l<CR>
: login 168302040.a a Tue  Nov 29 13:00:00 1983
: login 453400603.a a Mon Feb 27 08:15:00 1984
$
```

login will be your login name.

Try the following request. Using the **at** command, mail yourself a file at noon. The file, called *memo*, says that it is lunch time. You must redirect the file into **mail**. (You cannot type in the text directly unless you use the here document discussed in the *Shell Programming* section.) Now try the **at** command with the **−l** option.

```
$ at 12:00pm<CR>
mail mylogin < memo<CR>
<^d>
job 263131754.a at Jun 30 12:00:00 1985
$
$ at -l<CR>
: mylogin 263131754.a at Jun 30 12:00:00 1985
$
```

A brief recap of the **at** command follows next.

Command Recap

at - execute a list of commands at a specified time.

command	*options*	*arguments*
at		**time date**
	−r	**jobnumber**
	−l	

Description: Executes commands at the time specified. The order of the arguments is the time which can be 1 to 4 digits and "am" or "pm". The date need not be added if it is today. The date is specified by a month name followed by the number for the day.

Options: The **−r** option with the job number removes previously scheduled jobs.

The **−l** option without arguments gives the status of **at** and **batch** jobs and job numbers.

Remarks: Some times and dates are: at 08:15am Feb 27 *and* at 5:14pm Sept 24.

Obtaining the Status of Running Processes

The **ps** command will give you the status of the processes you are running.

Running a process or command in background with **&** was discussed in the section on special characters. The **ps** command will tell you the status of those processes. In the next example, the **grep** command was run in the background, and then the **ps** command was typed in. The system response, the output from the **ps** command, gives the PID, which is the process identification number, and TTY, which is the current number identification assigned to the terminal you are logged in on. It also gives the cumulative execution TIME for each process, and the COMMAND that is being executed. The PID is an important number if you decide to stop the execution of that command.

```
$grep word * &<CR>
28223
$
$ ps<CR>
PID      TTY TIME  COMMAND
28124    10  0:00  sh
28223    10  0:04  grep
28224    10  0:04  ps
$
```

The example not only gives you the PID for the **grep** command, but also for the other processes that are running, the **ps** command itself, and the **sh** command that is always running as long as you are logged in. **sh** is the shell program that interprets the shell commands. It is discussed in *Chapter 1* and *Chapter 4*.

Command Recap

ps - report process status

command	*options*	*arguments*
ps	several*	none

Description: Displays information about active processes.

Options: This command has several options. If you do not use any options you will get the status of the active shell processes that you are running.

Remarks: Gives you the PID, the Process ID. This is needed if you are going to **kill** the process, that is, stop the process from executing.

* See the *UNIX System User Reference Manual* for all available options and an explanation of their capabilities.

Terminating Active Processes

The **kill** command is used to stop active shell processes. The general format for the **kill** command is:

kill PID<CR>

What do you do if you decide you do not need to execute the command that you are running in the background?. If you press the BREAK key or the DEL key, you will find it does not stop the background process as it does the interactive commands. The **kill**

command terminates a background process. If you want to terminate the **grep** command used in the previous example:

```
$ kill 28223<CR>
28223 Terminated
$
```

A recap of the **kill** command follows.

Command Recap

kill - terminate a process

command	*options*	*arguments*
kill	available*	**job number or PID**

Description: **kill** will terminate the process given by the PID.

* See the *UNIX System User Reference Manual* for all available options and an explanation of their capabilities.

Using the No Hang Up Command

Another way to kill all processes is to hang up on the system, to log off. What if you want the background process to continue to run after you have logged off? The **nohup** command will allow background commands to continue to run even if you log off.

nohup command &<CR>

If you place the **nohup** command at the beginning of the command that you will be running as a background process, the background process will continue to run to completion after you have logged off.

Type in: **nohup grep word * > word.list &<CR>**

The **nohup** command can be stopped by the **kill** command. The recap of the **nohup** command is the following:

Command Recap

nohup - runs a command, ignoring hanging up or quitting the system

command	*options*	*arguments*
nohup	none	**command line**

Description: Executes a command line, even if you hang up or quit the system.

Now that you have mastered these shortcuts in the shell commands, use them in your shell programs.

COMMAND LANGUAGE EXERCISES

1-1. What happens if you use the * at the beginning of a file name? Try to list some of the files in a directory using the * with a last letter of one of your file names. What happens?

1-2. Try out the following two commands.

Type in: **cat [0-9]*<CR>**
echo *<CR>

1-3. Can you use **?** at the beginning or in the middle of a file name generation? Try it.

1-4. Do you have any files that begin with a number? Can you list them without listing the other files in your directory? Can you list only those files that begin with a lowercase letter between a and m? (Hint use a range of numbers or letters in [ ]).

1-5. Can you place a command in background mode on the line that is executing several other commands sequentially. Try it. What happens? (Hint use **;** and **&.**) Can the command in background mode be placed at any position on the command line? Try it. Experiment with each new character that you learn, so that you can learn the full power of the character.

1-6. Using the command line

cd; pwd; ls; ed trial<CR>

redirect the output of **pwd** and **ls** into a file. Remember, if you want to redirect both commands to the same file, you have to use >> for the second redirection or you will wipe out the information from the **pwd** command.

1-7. Instead of cutting the time out of the **date** response, try redirecting only the date, without the time, into **banner**. What is the only part that you need to change in the "time" command line?

banner `date | cut −c12-19`

SHELL PROGRAMMING

Getting Started

Let a shell program perform your tasks for you. A shell program is a UNIX system file that contains the commands that you would use to perform your task.

- How do you create a simple shell program?
- What makes the program run?
- Is there a special directory for your shell programs?

In this section of the tutorial you will learn the answers to these questions. The examples for creating shell programs usually show

two display screens. The first screen displays the contents of the file containing the commands used in your program. It shows the command line

cat file<CR>

and the system response to that command, which is the contents of the file.

```
$ cat file<CR>
First command
    .
    .
    .
Last command
$
```

The **$** indicates the shell prompt. The second screen shows the results of executing your shell program.

```
$ file<CR>
Results
$
```

The names of the file containing the shell program will be printed in **bold** in the text, since it is a command and not an ordinary text file.

Before you begin to create shell programs, you should be familiar with one of the editors. The editors are discussed in the tutorials in *Chapter 5* and *Chapter 6*.

Creating a Simple Shell Program

How do you think you would create a simple shell program that would:

- Tell you the directory you were in,
- List the contents of that directory, and then
- Display on your terminal: "This is the end of the shell program".

Think about it now before you read any further.

To create the shell program, you will need the following three shell commands:

pwd The command that prints the path name of the current directory,

ls The command that lists the contents of the current directory, and

echo The command that displays on your terminal the characters following **echo**.

To create your shell program, using **pwd**, **ls**, and **echo**, enter an editor and type in the following three commands.

Type in: **pwd<CR>**
ls<CR>
echo This is the end of the shell program.**<CR>**

Write the contents of the editor buffer to a file called **dl** (for directory list) and quit the editor. You have just created a shell program.

```
$ cat dl<CR>
pwd
ls
echo This is the end of the shell program.
$
```

Executing a Shell Program

How do you tell the shell that your file is a shell program that needs to be executed? The simplest way to execute a program is to use the **sh** command.

Type in: **sh dl<CR>**

What happened?

Did you notice the path name of the current directory printed out first, then the list of the contents of the current directory, and last of all the comment *This is the end of the shell program.* ?

The **sh** command is a good way to test out your shell program to make sure that it works.

If **dl** is a useful command, you will want to change the file permissions so that you need only type in **dl** to execute the command. The command that changes the permissions on a file, **chmod**, is discussed in *Chapter 3*. The example below reminds you how to type in the **chmod** command to make a file executable, and then do an **ls −l** so you can see the change in the permissions.

```
$ chmod u+x dl<CR>
$  ls -l<CR>
total 4
 -rw------- 1  login  login   3661  Nov  2  10:28 mbox
drwxrwxrwx 2  login  login   1056  Nov 11  18:20 rje
 -rwx------ 1  login  login     48  Nov 15  10:50 dl
$
```

Now you have an executable program **dl** in your current directory.

Type in: **dl<CR>**

Did the **dl** command execute?

Creating a bin Directory for Executable Files

If your shell program is useful, you will want to keep it in a special directory called *bin,* which is under your login directory.

If you want your **dl** command accessible from all your directories, make a *bin* directory from your login directory and move the **dl** file to your *bin*. Below is a reminder of those commands. In this example, **dl** is in the login directory.

Type in: **mkdir bin<CR>**
mv dl bin/dl<CR>

Move to the *bin* directory and type in the **ls −l** command. Does **dl** still have execute permission?

Now move to another directory other than the login directory.

Type in: **dl<CR>**

What happened?

A command recap of your new program **dl** follows.

Shell Program Recap

dl - display the directory path and directory contents

command	*arguments*
dl	none

Description: Displays the output of the shell command **pwd** and then lists the contents of the directory.

The *bin* is the best place to keep your executable shell programs. It is possible to give the *bin* directory another name, but you need to change the shell variable **PATH** to do so. The shell variables are discussed briefly in this chapter. For more advanced information read the document the *UNIX System Shell Commands and Programming*. (See *Appendix A*.)

Problem:
You can give your shell program file any appropriate file name. However, you should not name your program with the same name as a system command. The system will execute your command and not the command of the system.

If you had named your **dl** program **mv**, each time you tried to move a file, the system would not move your file. It would have executed your program to display the directory name and list the contents.

Problem:
Another problem would occur if you had named the **dl** file **ls**, and then tried to execute the file **ls**. You would create an infinite loop. After some time, the system would give you an error message:

Too many processes, cannot fork

What happened? You typed in your new command **ls**. The shell read the command **pwd** and executed that command. Then the shell read the command **ls** in your file and tried to execute your **ls** command. This formed an infinite loop:

```
$  < ls<
   pwd
   ls >
   echo  This is the end of the shell program
```

UNIX system designers wisely set a limit on how many times this infinite loop can execute. One way to keep this from happening is to give the path name for the system's **ls** command, **/bin/ls**.

The following **ls** shell program would work.

```
$ cat ls<CR>
pwd
/bin/ls
echo This is the end of the shell program
```

If you name your command **ls**, then you can only execute the system command with **/bin/ls**.

Variables

If you enjoyed sending the **banner** birthday greeting, you could make a shell program that would pipe the banner printout into the electronic **mail**. A good shell program would let you send to a different login each time you executed the program. The login would then be a variable. There are two ways you can specify a variable for a shell program:

- Positional parameters and
- Variables that you define.

Positional Parameters

A positional parameter is a variable that is found in a specified position in the command line of your shell program. Positional parameters are typed in after the command. They are strings of characters delimited by spaces, except for the last parameter, which is ended with **<CR>**. If **pp1** is the first positional parameter, **pp2** is the second positional parameter, and ... **pp9** is the ninth positional parameter, then the command line of your shell program called **shell.prog** will look like this.

shell.prog pp1 pp2 pp3 pp4 pp5 pp6 pp7 pp8 pp9<CR>

The shell program will take the first positional parameter **(pp1)** and substitute it in the shell program text for the characters **$1**. The second positional parameter **(pp2)** will be substituted for the characters **$2**. The ninth positional parameter **(pp9)**, of course, will be substituted for the characters **$9**.

If you want to see how the positional parameters are substituted into a program, try typing the following lines into a file called **pp** (positional parameters).

Type in:
```
echo The first positional parameter is: $1<CR>
echo The second positional parameter is: $2<CR>
echo The third positional parameter is: $3<CR>
echo The fourth positional parameter is: $4<CR>
```

First the **echo** command tells which parameter will be displayed and then displays the parameter. The next example shows the contents of the file **pp**.

```
$ cat pp<CR>
echo The first positional parameter is: $1
echo The second positional parameter is: $2
echo The third positional parameter is: $3
echo The fourth positional parameter is: $4
$
```

The following example shows the results of giving the four positional parameters one, two, three, and four to the shell program **pp**. Remember to change the mode of **pp** to be executable.

```
$ chmod u+x pp<CR>
$
$ pp one two three four<CR>
The first positional parameter is: one
The second positional parameter is: two
The third positional parameter is: three
The fourth positional parameter is: four
$
```

Now, return to creating your shell program for the banner birthday greeting. Call the file **bbday**. What command line would go into that file? Before you go on reading, try it.

Did you get the following?

```
$ cat bbday<CR>
banner happy birthday | mail $1
```

Try sending yourself a birthday greeting. If your login name is *slowmo*, then the command line would be:

```
$ bbday slowmo<CR>
you have mail
$
```

The following is a brief recap of the shell program command **bbday**.

Shell Program Recap

bbday - mail a banner birthday greeting

command	*arguments*
bbday	**login**

Description: **bbday** mails "happy birthday" in poster-sized letters to the specified login.

The **who** command will tell you every login that is currently using the system. How would you make a simple shell program called **whoson** that will tell you if a particular login is currently working on the system? You could try the following:

```
$ who | grep boss<CR>
boss     tty51    Nov 29 17:01
$
```

This command pipes the output of the **who** command into the **grep** command. The **grep** command is searching for the characters "boss". Since login *boss* is currently logged into the system, the shell will respond with:

boss tty51 Nov 29 17:01

If the only response is a prompt sign, then login *boss* is not currently on the system because the **grep** command found nothing. Create a **whoson** shell program.

Below are the ingredients for your shell program **whoson**.

who The shell command that lists everyone on the system,

grep The search command, and

$1 The first positional parameter for your shell program.

The **grep** command searches the output of the **who** command for the parameter designated in the program by **$1**. If it finds the login, it will display the line of information. If it does not find the login in the output from **who**, it will display your prompt.

Enter an editor and type the following command line into a file called **whoson**.

Type in: **who | grep $1<CR>**

Write the file, quit the editor, and change the mode of the file **whoson** to have execute permission.

Now try using your login as the positional parameter for the new program **whoson**. What was the system's response?

If your login name is *slowmo*, your new shell command line would look like:

```
$ whoson slowmo<CR>
slowmo     tty26        Jan 24 13:35
$
```

The first positional parameter is **slowmo**. The shell substitutes **slowmo** for the **$1** in your program.

who | grep slowmo<CR>

The following is a brief recap of the **whoson** command.

Shell Program Recap

whoson - display login information if user is logged in

command	*arguments*
whoson	**login**

Description: If a user is on the system, displays the user's login, the **TTY** number, the time and date the user logged in.

The shell command line will allow 128 positional parameters. However, your shell program text is restricted to **$1** through **$9**, unless you use the **$*** described below, or the **shift** command, which is described in the document the *UNIX System Shell Commands and Programming*. (See *Appendix A*.)

Parameters with Special Meaning

$# This variable in your shell program will record and display the number of positional parameters you typed in for your shell program.

Let's look at an example that will show you what happens when you use **$#**. Put the following command lines in a shell program called **get.num**.

```
$ cat get.num<CR>
echo The number of parameters is: $#
$
```

The program counts all the positional parameters and displays that number. Give **get.num** four parameters. They can be any string of characters.

```
$ get.num test out this program<CR>
The number of parameters is: 4
$
```

Shell Program Recap

get.num - count and display the number of arguments

command	*arguments*
get.num	(any string)

Description: **get.num** counts the number of arguments given to the command and then displays that number.

Remarks: This command demonstrates the special parameter *$#*.

$*** This variable in your shell program will substitute all positional parameters starting with the first positional parameter. The parameter **$* does not restrict you to nine parameters.

You can make a simple shell program to demonstrate $*. Make a shell program called **show.param** that will **echo** all of the parameters. Type in the **echo** command line shown in the following screen.

```
$ cat show.param<CR>
echo The parameters for this command are: $*
$
```

Make **show.param** executable and try it out.

```
$ show.param hello how are you<CR>
The parameters for this command are: hello how are you
$
```

Now try **show.param** using more than nine positional parameters.

```
$ show.param one two 3 4 5 six 7 8 9 10 11<CR>
The parameters for this command are: one two 3 4 5 six
7 8 9 10 11
$
```

The **$*** is very handy if file generation names are used as the parameters.

Try a file name generation parameter in your **show.param** command. If you have several chapters of a manual in your directory called chap1, chap2 through chap7, you will get a printout listing of all of those chapters.

```
$ show.param chap?<CR>
The parameters for this command are: chap1 chap2 chap3
chap4 chap5 chap6 chap7
$
```

A quick recap of **show.param** follows.

Shell Program Recap

show.param - display all of the parameters

command	*arguments*
show.param	(any positional parameters)

Description: **show.param** displays all of the parameters.

Remarks: If the parameters are file name generations, it will display each of those file names.

You may want to practice with positional parameters so that they are familiar to you before you continue on to the next section in which you will name the variables within the program, rather than use them as arguments in a command line.

Variable Names

The shell allows you to name the variables within a shell program. Naming the variables in a shell program makes it easier for another person to use. Instead of using positional parameters, you will tell the user what to type in for the variable, or you will give the variable a value that is the output of a command.

What does a named variable look like? In the example below, **var1** is the name of the variable and **myname** is the value or character string assigned to that variable. There are no spaces on either side of the = sign.

var1=myname<CR>

Within the shell program, a **$** in front of the variable name alerts the shell that a substitution is needed in the shell program. **$var1** tells the shell to substitute the value **myname**, which was given to **var1**, for the characters **$var1**.

The first character of a variable name must be a letter or an underscore. The rest of the name can be composed of letters, underscores, and digits. As in the case of shell program file names, it is a risky business to use a shell command as a variable name. Also, the shell has reserved some variable names to be used by the shell. The following names are used by the shell and should not be used as the name of one of your variables. A brief explanation of each variable is given.

CDPATH
: This variable defines the search path for the **cd** command.

HOME
: This is the default variable for the **cd** command (Home Directory).

IFS
: This variable defines the internal field separators, normally the space, the tab, and the carriage return.

MAIL
: This variable is set to the name of the file that contains your electronic mail.

PATH
: This variable determines the path that is followed to find commands.

PS1
PS2
These variables define the primary and secondary prompt strings. The defaults are **$** and >. Do you have a prompt sign **$** ?

TERM
This variable tells the shell what kind of terminal you are working on. It is important to set this variable if you are editing with **vi**.

Many of these named variables are explained in the last section of this chapter on your login environment.

Assign Values to Variables

If you edit with **vi**, you know that you must set the variable TERM to equal the code for your type of terminal before you use the **vi** editor. For example:

TERM=T3<CR>

This is the simplest way to assign a value to a variable.

There are several other ways to assign values to variables. One way is to use the **read** command to assign input to the variable. Another way is to assign the value from the output of a command using back quotes ` ... `. A third way would be to assign a positional parameter to the variable.

Assign Values by the Read Command

You can set up your program so that you can type in the command and then be prompted by the program to type in the value for the variable. The **read** command assigns the input to the specified variable. The general format for the **read** command is:

read var<CR>

The values assigned by **read** to **var** will be substituted for **$var** in the program. If the **echo** command is executed just before the **read**

command, the program can display the directions "type in ...". The **read** command will wait until you type in the value, and then assign the string of characters that you type in as the value for the variable.

If you had a list that contained the names and telephone numbers of people you called often, you could make a simple shell program that would automatically give you someone's number. Stop for a minute. How would you make up the program using the following ingredients?

echo The command that echoes the instructions.

read The command that assigns the input value to the variable **name**.

grep The command that searches for the person's name and number.

First, you would use the **echo** command to inform the user to type in the name of the person to be called.

echo Type in the last name<CR>

The **read** command will then assign the person's name to the variable **name**.

read name<CR>

Notice that you do not use the = to assign the variable, the **read** command automatically assigns the typed in characters to **name**.

The **grep** command will then search your phone list for the name. If your phone list were called *list*, the command line would be:

grep $name list<CR>

In the next example, the shell program is called **num.please**. Remember, the system response to the **cat** command is the contents of the shell program file.

```
$ cat num.please<CR>
echo Type in the last name
read name
grep $name list
$
```

Make a list of last names and phone numbers and try **num.please**. Or, try the next example, which is a program that creates a list. You can use several variables in one program. If you have a phone list, you may want a quick and easy way to add names and numbers to the list. The program:

- Asks for the name of the person,
- Assigns the name to the variable **name**,
- Asks for the person's number,
- Assigns the number to the variable **num**, and
- Echos the **name** and **num** into the file *list*. You must use >> to redirect the output of the **echo** command to the bottom of your list. If you use >, your list will contain only the last phone number.

The program is called **mknum**.

```
$ cat mknum<CR>
echo Type in name
read name
echo Type in number
read num
echo $name $num >> list
$
$ chmod u+x mknum<CR>
$
```

Now try out the new programs for your phone list. In the next example, **mknum** creates the new listing for Mr. Niceguy. Then, **num.please** gives you Mr. Niceguy's phone number.

```
$ mknum<CR>
Type in the name
Mr. Niceguy<CR>
Type in the number
668-0007<CR>
$
$ num.please<CR>
Type in last name
Niceguy<CR>
Mr. Niceguy 668-0007
$
```

Notice that the variable **name** accepts both **Mr.** and **Niceguy** as the value.

Here is a brief recap of **mknum** and **num.please**.

Shell Program Recap

mknum - place name and number on a phone list

command	*arguments*
mknum	(interactive)

Description: Asks you for the name and number of a person and adds the name and number to your phone list.

Remarks: This is an interactive command.

Shell Program Recap

num.please - display a person's name and number

command	*arguments*
num.please	(interactive)

Description: Asks you for a person's last name, and then displays the name and telephone number.

Remarks: This is an interactive command.

Substitute Command Output for the Value of a Variable

Another way to assign a value to a variable is to substitute the output of a command for the value. This will be very useful in the next section when you try loops and conditional constructs.

The general format to assign output as the value for a variable is:

var=`command`<CR>

The variable **var** has the value of the output from **command**.

In one of the previous examples on piping, the **date** command was piped into the **cut** command to get the correct time. That command line was:

date | cut −c12-19<CR>

You can place that command in a simple shell program called **t** that will give you the time.

```
$ cat t<CR>
time=`date | cut -c12-19`
echo The time is: $time
$
```

Remember there are no spaces on either side of the equal sign.

Change the mode on the file and you now have a program that gives you the time.

```
$ chmod u+x t<CR>
$ t<CR>
The time is: 10:36
$
```

The recap for the **t** shell program follows.

Shell Program Recap

t - display the correct time

command	*arguments*
t	none

Description: **t** gives you the correct time in hours and minutes.

Assign Values with Positional Parameters

A positional parameter can be assigned to a named parameter. For example:

var1=$1<CR>

The example below is a simple program **simp.p** that demonstrates how you can assign a positional parameter to a variable. The command lines in the file would be the following:

```
$ cat simp.p<CR>
var1=$1
echo $var1
$
```

Or, you can assign the output of a command that uses a positional parameter.

person=`who | grep $1`<CR>

If you wanted to keep track of the results of your **whoson** program, you could create the program **log.time**. The output of your **whoson** shell program is assigned to the variable **person**. Then, that value **$person** is added to the file *login.file* with the **echo** command. The last part of the program displays the value of **$person**, which is the same as the response to the **whoson** command.

```
$ cat log.time<CR>
person=`who | grep $1`
echo $person >> login.file
echo $person
$
```

The system response to **log.time** would appear as in the following screen.

```
$ log.time maryann<CR>
maryann      tty61          Apr 11 10:26
$
```

The following is a quick recap of the **log.time** program.

Shell Program Recap

log.time - log and display a specified login that is currently logged in

command	*arguments*
log.time	**login**

Description: If the specified login is currently on the system, **log.time** places the line of information from the **who** command into the file *login.file* and then displays that line of information on your terminal.

As you do more programming, you may discover other ways to assign variables that will help you in shell programs.

Shell Programming Constructs

The shell programming language has several constructs that give you more flexibility in your programs.

- The "here document" allows you to redirect lines of input into a command.
- The looping constructs **for** or **while** cause a program to reiterate commands in a loop.

- The conditional control commands, **if** or **case**, execute a group of commands only if a particular set of conditions is met.
- The **break** command gives the unconditional end of a loop.

Comments

Before you begin writing shell programs with loops, you may want to know how to put comments about your program into the file, which the system will ignore. To place comments in a program, begin the comment with # and end it with **<CR>**. The general format for a comment line is:

#comment<CR>

The shell will ignore all characters after the #. These lines

 # This program sends a generic birthday greeting<CR>
 # This program needs a login as the positional parameter<CR>

will be ignored by the system when your program is being executed. They only serve as a reminder to you, the programmer.

The Here Document

The here document allows you to redirect lines of input of a shell program into a command. The here document consists of the redirection symbol << and the delimiter that specifies the beginning and end of the lines of input. The delimiter can be one character or a string of characters. The **!** is often used as a delimiter. The general format for the here document is:

```
command <<!<CR>
...input lines...<CR>
!<CR>
```

The here document could be used in a shell program, to redirect lines of input into the **mail** command. The program shown below sends a generic birthday greeting with the **mail** command. The program is called **gbday**.

```
$ cat gbday<CR>
mail $1 <<!
Best wishes to you on your birthday.
!
$
```

The person's login is the first positional parameter **$1**.

The redirected input is:

Best wishes to you on your birthday.

To send the greeting:

```
$ gbday mary<CR>
$
```

To receive the greeting, login *mary* would execute the **mail** command.

```
$ mail<CR>
From mylogin Mon May 14 14:31 CDT 1984
Best wishes to you on your birthday
$
```

The following is a recap of **gbday**.

Shell Program Recap

gbday - send a generic birthday greeting

command	*arguments*
gbday	**login**

Description: **gbday** sends a generic birthday greeting to the login given as an argument.

Using ed in a Shell Program

The line editor **ed** can be used within a shell program if it is combined with the here document commands.

Suppose you want to make a shell program that will enter the editor, **ed**, make a global substitution to a file, write the file, and then quit the editor. The **ed** command to make a global substitution is:

g/text to be changed/s//new text/g<CR>

Before you read any further, jot down what you think the command sequence will be. Put your command sequence into a file called **ch.text**. If you want to suppress the character count of **ed** so that it will not appear on your terminal, use the – option:

ed – filename<CR>

Try to execute the file. Did it work?

If you used the **read** command to enter the variables, your program **ch.text** may look similar to what appears in the following screen.

```
$ cat ch.text<CR>
echo Type in the file name.
read file1
echo Type in the exact text to be changed.
read oldtext
echo Type in the exact new text to replace the above.
read newtext
ed - $file1 <<!
g/$oldtext/s//$newtext/g
w
q
!
$
```

This program uses three variables. Each of them is entered into the program with the **read** command.

$file The name of the file to be edited.

$oldtext The exact text to be changed.

$newtext The new text.

Once the variables are entered into the program, the here document redirects the global, write, and quit commands into the **ed** command. Try out the new **ch.text** command.

```
$ ch.text<CR>
Type in the filename.
memo<CR>
Type in the exact text to be changed.
Dear John:<CR>
Type in the exact new text to replace the above.
To whom it may concern:<CR>
$ cat memo<CR>
To whom it may concern:
$
```

Did you try to use positional parameters? Did you have any problems entering the text changes as variables, or did you quote the character strings for each parameter?

The recap of the **ch.text** command is:

Shell Program Recap

ch.text - change text in a file

command	*arguments*
ch.text	(interactive)

Description: Replaces text in a file with new text.

Remarks: This shell program is interactive. It will prompt you to type in the arguments.

If you want to become more familiar with the line editor **ed**, see *Chapter 5, Line Editor Tutorial (ed).*

The stream editor **sed** can also be used in shell programming. More information on that editor can be found in the *UNIX System Editing Guide.* (See *Appendix A.*)

Looping

Until now, the commands in your shell program have been executed once and only once and in sequence. Looping constructs give you repetitive execution of a command or group of commands. The **for** or **while** commands will cause a program to loop and execute a sequence of commands several times.

The for Loop

The **for** loop executes a sequence of commands for each member of a list. The **for** command loop also requires the keywords **in, do,** and **done**. The **for, do,** and **done** keywords must be the first word on a line. The general format of the **for** loop is:

```
for variable<CR>
      in this list of values<CR>
do the following commands<CR>
     command 1<CR>
     command 2<CR>
          .
          .
          .
     last command<CR>
done<CR>
```

The variable can be any name you choose. If it is **var**, then the values given after the keyword **in** will be sequentially substituted for **$var** in the command list. If **in** is omitted, the values for **var** will be the positional parameters. The command list between the keywords **do** and **done** will be executed for each value.

When the commands have been executed for the last value, the program will execute the next line below **done**. If there is no line, the program will end.

It is easier to read a shell program if the looping constructs stand out. Since the shell ignores spaces at the beginning of the lines, each section of commands can be indented as it was in the above format. Also, if you indent each command section, you can quickly check to make sure each **do** has a corresponding **done** statement to end the loop.

The easiest way to understand a shell programming construct is to try an example. Try to create a program that will move files to another directory.

The ingredients for the program are:

echo	You want to echo directions to type in the path name to reach the new directory.
read	You want to type in the path name, and assign it to the variable **path**.
for variable	You must name the variable. Call it **file** for your shell program. It will appear as **$file** in the command sequence.
in list of values	The list of values will be the file names. If the **in** clause is omitted, the list of values is taken to be **$***, that is, the parameters entered on the command line.
do command sequence	The command sequence for this program is: **mv $file $path/$file<CR>**
done	

Your shell program text for the program called **mv.file** will look like:

```
$ cat mv.file<CR>
echo Please type in the directory path
read path
for file
    in memo1 memo2 memo3
do
    mv $file $path/$file
done
$
```

Notice that you did not type in any values for the variable **file**. The values are already in your program. If you want to change the files each time you invoke the program, use positional parameters or

variables that you name. You do not need the **in** keyword to list the values when you use positional parameters. If you choose positional parameters, your shell program will look like:

```
$ cat mv.file<CR>
echo type in the directory path
read path
for file
do
    mv $file $path/$file
done
$
```

It is likely that you will want to move several files using the special file name generation characters.

If this is a useful command, remember to move it into your *bin*.

Following is a recap of the **mv.file** shell program.

Shell Program Recap

mv.file - move files to another directory

command	*arguments*
mv.file	file names (interactive)

Description: Moves files to a directory.

Remarks: This program requires the file names to be given as positional parameters. The path to the new directory is asked for interactively by the program.

The while Loop

The **while** loop will continue executing the sequence of commands in the **do...done** list as long as the final command in the **while** command list returns a status of true, that is can be executed. The **while, do,** and **done** keywords must be the first characters on the line. The general format of the **while** loop is the following:

```
while<CR>
        command 1<CR>
            .
            .
            .
        last command<CR>
do<CR>
        command 1<CR>
            .
            .
            .
        last command<CR>
done<CR>
```

A simple program using the **while** loop enters a list of names into a file. The command lines for that program called **enter.name** are:

```
$ cat enter.name<CR>
while
    read x
do
    echo $x>>xfile
done
$
```

This shell program needs some instructions. You have to know to delimit or separate the names by a **<CR>**, and you have to use a **<^d>** to end the program. Also, it would be nice if your program

displayed the list of names in the *xfile* at the end of the program. If you added those ingredients to the program, the commands lines for the program become:

```
$ cat enter.name<CR>
echo 'Please type in each person's name and then a <CR>'
echo 'Please end the list of names with a <^d>'
while read x
do
    echo $x>>xfile
done
echo xfile contains the following names:
cat xfile
$
```

Notice that after the loop is completed, the program executes the commands below the **done**.

In the **echo** command line, you used characters that are special to the shell, so you must use the '...' to turn off that special meaning. Put the above command lines in an executable file and try out the shell program.

```
$ enter.name<CR>
Please type in each person's name and then a <CR>
Please end the list of names with a <^d>
Mary Lou<CR>
Janice<CR>
<^d>
xfile contains the following names:
Mary Lou
Janice
$
```

Conditional Constructs if...then

The **if** command tells the shell program to execute the **then** sequence of commands only if the final command in the **if** command list is successful. The **if** construct ends with the keyword **fi**. The general format for the **if** construct is as follows:

```
if<CR>
      command1<CR>
            .
            .
            .
      last command<CR>
   then<CR>
         command1<CR>
               .
               .
               .
         last  command<CR>
fi<CR>
```

The next shell program demonstrates the **if...then** construct. The program will search for a word in a file. If the **grep** command is successful then the program will **echo** that the word is found in the file. In this example the variables are read into the shell program. Type in the shell program shown below and try it out. Call the program **search**.

```
$ cat search<CR>
echo Type in the word and the file name.
read word file
if grep $word $file
   then echo $word is in $file
fi
$
```

Notice that the **read** command is assigning values to two variables. The first characters that you type in, up to a space, are assigned to **word**. All of the rest of the characters including spaces will be assigned to **file**.

Pick a word that you know is in one of your files and try out this shell program. Did you see that even though the program works, there is an irritating problem? Your program displayed more than the line of text called for by the program. The extra lines of text displayed on your terminal were the output of the **grep** command.

The Shell Garbage Can /dev/null

The shell has a file that acts like a garbage can. You can deposit any unwanted output in the file called */dev/null*, by redirecting the command output to */dev/null*.

Try out */dev/null* by throwing out the results of the **who** command. First, type in the **who** command. The response tells you who is on the system. Now, try the **who** command, but redirect the response into the file */dev/null*.

who > /dev/null<CR>

The response displayed on your terminal was your prompt. The response to the **who** command was placed in */dev/null* and became null, or nothing. If you want to dispose of the **grep** command response in your **search** program, change the **if** command line.

if grep $word $file > /dev/null<CR>

Now execute your **search** program. The program should only respond with the text of the **echo** command line.

The **if...then** construction can also issue an alternate set of commands with **else**, when the **if** command sequence is false. The general format of the **if...then...else** construct follows.

```
if<CR>
      command1<CR>
            .
            .
            .
      last command<CR>
   then<CR>
         command1<CR>
               .
               .
               .
         last command<CR>
   else<CR>
         command1<CR>
               .
               .
               .
         last command<CR>
fi<CR>
```

You can now improve your **search** command. The shell program **search** can look for a word in a file. If the word is found, the program will tell you the word is found. If it is not found (**else**) the program will tell you the word was NOT found. The text of your **search** file will look like the following:

```
$ cat search<CR>
echo Type in the word and the file name.
read word file
if
   grep $word $file >/dev/null
then
   echo $word is in $file
else
   echo $word is NOT in $file
fi
$
```

Following is a quick recap of the enhanced shell program called **search**.

Shell Program Recap

search - tell if a word is in a file

command	*arguments*
search	interactive

Description: Tells the user whether or not a word is in a file.

Remarks: The arguments, the word and the file, are asked for interactively.

The test Command for Loops

test is a very useful command for conditional constructs. The **test** command checks to see if certain conditions are true. If the condition is true, then the loop will continue. If the condition is false, then the loop will end and the next command is executed. Some of the useful options for the **test** command are:

test −r filename<CR>
True if the file exists and is readable

test −w filename<CR>
True if the file exists and has write permission

test −x filename<CR>
True if the file exists and is executable

test −s filename<CR>
True if the file exists and has at least one character

If you have not changed the values of the **PATH** variable that were initially given to you by the system, then the executable files in your *bin* directory can be executed from any one of your directories. You may want to create a shell program that will move all the executable files in the current directory to your *bin* directory. The **test −x**

command can be used to select the executable files from the list of files in the current directory. Review the **mv.file** program example of the **for** construct.

```
$ cat mv.file<CR>
echo type in the directory path
read path
for file
do
  mv $file $path/$file
done
$
```

Include an **if test −x** statement in the **do...done** loop to move only those files that are executable.

If you name the shell program **mv.ex**, then the shell program will be as follows:

```
$ cat mv.ex<CR>
echo type in the directory path
read path
for file
  do
    if test −x $file
        then
          mv $file $path/$file
    fi
  done
$
```

The directory path will be the path from the current directory to the *bin* directory. However, if you use the value for the shell variable **HOME**, you will not need to type in the path each time. **$HOME** gives the path to the login directory. **$HOME/bin** gives the path to your *bin*.

```
$ cat mv.ex<CR>
for file
  do
    if test -x $file
       then
          mv $file $HOME/bin/$file
    fi
  done
$
```

To execute the command, use all the files in the current directory, * , as the positional parameters. The following screen executes the command from the current directory and then moves to the *bin* directory and lists the files in that directory. All the executable files should be there.

```
$ mv.ex *<CR>
$ cd; cd bin; ls<CR>
```

Shell Program Recap

mv.ex - move all executable files in the current directory to the *bin* directory

command	*arguments*
mv.ex	**all file names (*)**

Description: Moves all the files with execute permission that are in the current directory to the *bin* directory

Remarks: All executable files in the *bin* directory (or the directory indicated by the **PATH** variable) can be executed from any of your directories.

The Conditional Construct case..esac

The **case...esac** is a multiple choice construction that allows you to choose one of several patterns and then execute a list of commands for that pattern. The keyword **in** must begin the pattern statements with their command sequence. You must place a **)** after the last character of each pattern. The command sequence for each pattern is ended with ;;. The **case** construction must be ended with **esac** (letters of case reversed). The general format for the **case** construction is:

```
case characters<CR>
in<CR>
    pattern1)<CR>
        command line 1<CR>
            .
            .
            .
        last command line<CR>
    ;;<CR>
    pattern2)<CR>
        command line 1<CR>
            .
            .
            .
        last command line<CR>
    ;;<CR>
    pattern3)<CR>
        command line 1<CR>
            .
            .
            .
        last command line<CR>
    ;;<CR>
    *)<CR>
        command 1<CR>
            .
            .
            .
        last command<CR>
    ;;<CR>
esac<CR>
```

The **case** construction will try to match characters with the first pattern. If there is a match, the program will execute the command lines after the first pattern and up to the ;;.

If the first pattern is not matched, then the program will proceed to the second pattern. After a pattern is matched, the program does not try to match any more of the patterns, but goes to the command following **esac**. The * used as a pattern at the end of the list of patterns allows you to give instructions if none of the patterns are matched. The * means any pattern, so it must be placed at the end of the pattern list if the other patterns are to be checked first.

If you have used the **vi** editor, you know you must assign a value to the **TERM** variable so that the shell knows what kind of terminal is going to display the editing window of **vi**. A good example of the **case** construction would be a program that will set up the shell variable **TERM** for you according to what type of terminal you are logged in on. If you log in on different types of terminals, the program **set.term** will be very handy for you.

set.term will ask you to type in the terminal type, then it will set **TERM** equal to the terminal code. You may want to glance back at the beginning of the **vi** tutorial for the explanation of those commands. The command lines are:

```
TERM=terminal code<CR>
export TERM<CR>
```

In this example of **set.term**, the person uses either a TELETYPE 4420, TELETYPE 5410, or a TELETYPE 5420.

The **set.term** program will first check if the value of **term** is 4420. If it is, then it will assign the value T4 to **TERM**, and exit the program. If it is not 4420, it will check for 5410 and then for 5420. It will execute the commands under the first pattern that it finds, and then go to the next command after the **esac** command.

At the end of the patterns for the TELETYPE terminals, the pattern * , meaning everything else, will warn you that you do not have a pattern for that terminal, and it will also allow you to leave the **case** construct.

```
$ cat set.term<CR>
echo If you have a TTY 4420 type in 4420
echo If you have a TTY 5410 type in 5410
echo If you have a TTY 5420 type in 5420
read term
case $term
   in
        4420)
             TERM=T4
        ;;
        5410)
             TERM=T5
        ;;
        5420)
             TERM=T7
        ;;
        *)
             echo not a correct terminal type
        ;;
esac
export TERM
echo end of program
$
```

What would have happened if you had placed the * pattern first? The **set.term** program would never assign a value to TERM since it would always fit the first pattern *, which means everything.

When you read the section on modifying your login environment, you may want to put the **set.term** command in your *bin*, and add the command line

set.term<CR>

to your *.profile*.

Following is a quick recap of the **set.term** shell program.

Shell Program Recap

set.term - assign a value to TERM

command	*arguments*
set.term	interactive

Description: Assigns a value to the shell variable TERM and then exports that value to other shell procedures.

Remarks: This command asks for a specific terminal code to be used as a pattern for the **case** construction.

Unconditional Control Statement break

The **break** command unconditionally stops the execution of any loop in which it is encountered, and goes to the next command after the **done, fi**, or **esac** statement. If there are no commands after that statement, the program ends.

In the example for the program **set.term**, the **break** command could have been used instead of the **echo** command.

```
$ cat set.term<CR>
echo If you have a TTY 4420 type in 4420
echo If you have a TTY 5410 type in 5410
echo If you have a TTY 5420 type in 5420
read term
case $term
   in
        4420)
        TERM=T4
   ;;
   5410)
        TERM=T5
   ;;
   5420)
        TERM=T7
   ;;
   *)
        break
   ;;
esac
export TERM
echo end of program
$
```

As you do more shell programming, you may want to use two other unconditional commands, the **continue** command and the **exit** command. The **continue** command causes the program to go immediately to the next iteration of a **do** or **for** loop without executing the remaining commands in the loop.

Normally, a shell program terminates when the end of the file is reached. If you want the program to end at some other point, you can use the **exit** command. Both of these commands are explained in detail in the *UNIX System Shell Commands and Programming*. (See *Appendix A*.)

Debugging Programs

Debugging is computer slang for finding and correcting errors in a program. There will be times when you will execute a shell program and nothing will happen. There is a "bug" in your program.

Your program may consist of several steps or several groups of commands. How do you discover which step is the culprit? There are two options to the **sh** command that will help you debug a program.

sh −v<CR>	Prints the shell input lines as they are read by the system.
sh −x<CR>	Prints commands and their arguments as they are executed.

To try out these two options, create a shell program that has an error in it. For example, type in the following list of commands in a file called **bug**.

```
$ cat bug<CR>
today=`date`
person=$1
mail $2
$person
When you log off come into my office please.
$today.
MLH
$
```

The mail message sent to Tom ($1) at login *tommy* ($2) should read as shown in the following screen.

```
$ mail<CR>
From mlh  Thu  Apr 10  11:36  CST  1984
Tom
When you log off come into my office please.
Thu  Apr 10  11:36:32  CST  1984
MLH
$
?
.
```

If you try to execute **bug**, you will have to press the BREAK key or the DEL key to end the program.

To debug this program, try **sh −v**, which will print the lines of the file as they are read by the system.

```
$ sh −v bug tom tommy<CR>
today=`date`
person=$1
mail $2
```

Notice that the output stops on the **mail** command. There is a problem with **mail**. The here document must be used to redirect input into **mail**.

Before you fix the **bug** program, try **sh −x**, which prints the commands and their arguments as they are read by the system.

```
$ sh -x bug tom tommy<CR>
+date
today=Thu  Apr 10  11:07:23  CST  1984
person=tom
+ mail tommy
```

Once again, the program stops at the **mail** command. Notice that the substitutions for the variables have been made and are displayed.

The corrected **bug** program is as follows:

```
$ cat bug<CR>
today=`date`
person=$1
mail $2 <<!
$person
When you log off come into my office please.
$today
MLH
!
$
```

The **tee** command is a helpful command to debug pipe lines. It places a copy of the output of a command into a file that you name, as well as piping it to another command. The general form of the **tee** command is:

command1 | tee save.file | command2<CR>

save.file is the name of the file that will save the output of **command1** for you to study.

If you wanted to check on the output of the **grep** command in the following command line

who | grep $1 | cut −c1-9<CR>

you can use **tee** to copy the output of **grep** into a file to check after the program is done executing.

who | grep $1 | tee check | cut −c1-9<CR>

The file *check* contains a copy of the output from the **grep** command.

```
$ who | grep mlhmo | tee check | cut −c1-9<CR>
$ mlhmo
$ cat check<CR>
mlhmo    tty61    Apr 10    11:30
$
```

If you do a lot of shell programming, you will want to refer to the *UNIX System Shell Commands and Programming* and learn about command return codes and redirecting standard error.

Modifying Your Login Environment

What is a .profile?

When you log in, the shell first looks at a file in your login directory called the *.profile* (pronounced "dot profile"). The *.profile* is a shell program that issues commands to control your shell environment.

Since the *.profile* is a file, it can be edited and changed to suit your needs. On some systems you can edit this file yourself, and on other systems the system administrator will do this for you.

If you can edit the file yourself, you may want to be cautious the first few times and make a copy of your *.profile* in another file called *safe.profile*.

```
$ cp .profile safe.profile<CR>
$
```

You can add commands to your *.profile* just as you can add commands to any other shell program. You can also set some terminal options with the **stty** command, and you can set some shell variables.

Adding Commands to .profile

How do you add commands to your *.profile*? Try this pleasant example. The UNIX system will allow you to start out your day with a message from your computer. Edit your *.profile* and add the following **echo** command to the last line of the file.

Type in:

echo Good Morning! I am ready to work for you.

Write and quit the editor.

Whenever you make changes to your *.profile* and you want to initiate them in the current work session, you may type in a . and space before *.profile*. The shell will reinitialize your environment, that is, it will read and execute the commands in your *.profile*.

Now, experience communicating with your computer.

Type in: **. .profile<CR>**

The system should respond with:

Good Morning! I am ready to work for you
$

Setting Terminal Options

The **stty** command can make your shell environment more convenient for you. You can set the following options for **stty**.

stty −tabs

This option preserves tabs when you are printing. It expands the tab setting to eight spaces, which is the default. The number of spaces for each tab can be changed. Read the *UNIX System User Reference Manual* on **stty** for more details.

stty erase <^h>

This option allows you to use the erase key on your keyboard to erase a letter, instead of the default character #. Usually this key is the BACK SPACE key.

stty echoe

If you have a terminal with a screen, this option erases characters from the screen as you erase them with the BACK SPACE key.

If you want to use these options for the **stty** command, you create those command lines in your *.profile* just as you would create them in one of your the shell programs. If you use the **tail** command, which displays the last few lines of a file, you can see the results of adding those four command lines to your *.profile*.

```
$ tail −4 .profile<CR>
echo Good Morning! I am ready to work for you
stty −tabs
stty erase <^h>
stty echoe
$
```

If you have not used the **tail** command before, the following is a brief recap of **tail**.

Command Recap

tail - display the last portion of a file

command	*options*	*arguments*
tail	*-n*	**file name**

Description: Displays the last lines of a file.

Options: Using the option you can specify number of lines *n*. The default (no options) is ten lines. There are other options, besides specifying *-n*. You can specify blocks (b) or characters (c) instead of lines.

Using Shell Variables

Several of the variables reserved by the shell are used in your *.profile*.

Let's take a quick look at four of these variables.

HOME

This variable gives the path for your login directory. Go to your login directory and type in **pwd<CR>**. What was the system response? Now type in **echo $HOME<CR>**. Was the system response the same as the response to **pwd**? **$HOME** is the default option for **cd**. If you do not specify a directory for **cd**, it will move you to **$HOME**.

PATH

This variable gives the system the search path for finding and executing commands.

If you want to see the current values for your **PATH** variable type in: **echo $PATH**.

```
$ echo $PATH<CR>
:/mylogin/bin:/bin:/usr/bin:/usr/lib
$
```

The : is a delimiter. Notice that for this **PATH** the system looks in */mylogin/bin,* for the command first, then into */bin,* then into */usr/bin,* and so on.

If you are working on a project with several other people, you may want to set up a group *bin,* a directory of special shell programs used only by your group. The directory would be found from the root directory. The path would be */group/bin.* How do you add this to your **PATH** variable? Edit your *.profile,* and add *:/group/bin* to the end of your **PATH**.

PATH=:/mylogin/bin:/bin:/usr/lib:/group/bin<CR>

TERM

This variable tells the shell what kind of terminal you are working on. If you have done any editing with **vi** you know that you have to specify:

TERM=code<CR>
export TERM<CR>

Not only do you have to tell the shell what kind of terminal you are working on but you must **export** the variable. If you read the *UNIX System Shell Commands and Programming,* you will learn why variables need to be exported.

If you do not want to specify the **TERM** variable each time you log in you can add those two command lines to your *.profile* and they will automatically be recognized each time you log into the UNIX system. Or, if you log in on more than one type of terminal, you will want to add your **set.term** command to your *.profile.*

PS1

One of the delightful things about your *.profile* is that you can change your prompt. This one is fun to experiment with. Try the following example. If you wish to use several words, remember to quote the phrase. Also, if you use quotes you can add a carriage return to your prompt.

Type in: **PS1="Your wish is my command<CR>"**

Now your prompt sign looks like:

```
$ . .profile<CR>
Your wish is my command
```

The mundane **$** is gone forever, or until you delete the PS1 variable from your *.profile*.

Conclusion

This tutorial has given you the basics for creating some shell programs. If you have logged in and tried the examples and exercises as you read the tutorial, you can probably perform many of your day-to-day tasks with your new shell programs. Shell programming can be much more complex and perform more complicated tasks than shown in this brief tutorial. If you want to read further on shell commands and programming, read the *UNIX System User Reference Manual* on the **sh** command, and read the *UNIX System Shell Commands and Programming*.

SHELL PROGRAMMING EXERCISES

2-1. Make the command line

banner `date | cut −c12-19`<CR>

into a shell program called **time**.

2-2. Make a shell program that will give only the date in a banner display. Be careful what you name the program!

2-3. Make a shell program that will send a note to several people on your system.

2-4. Redirect the date command without the time into a file.

2-5. Echo the phrase "Dear colleague" in the same file as the date command without erasing the date.

2-6. Using the above exercises, make a shell program that will send a memo with:

- Current date and the "Dear colleague" at the top of the memo,
- Body of a file that is the memo, and
- Closing statement

to the same people on your system as in Exercise 2-3.

2-7. How would you **read** variables into the **mv.file** program.

2-8. Use the **for** loop to move a list of files in the current directory to another directory.

How would you move all files to another directory?

Ingredients:
*
$*
mv $file newdirectory

2-9. How would you change the program **search**, to search through several files?

Hint:
for file
in $*

2-10. Set the **stty** options for your environment.

2-11. Give yourself a new prompt that includes a carriage return. (Hint " **<CR>**")

2-12. Check to see what $HOME, $TERM, and $PATH are set to in your environment.

ANSWERS TO EXERCISES

Command Language Exercises

1-1. The * at the beginning of a file name will refer to all files that end in that file name, including that file name.

```
$ls *t<CR>
cat
123t
new.t
t
$
```

1-2. **cat [0-9]*** would display the files:

1memo
100data
9
05name

echo * will list all the files in the current directory.

1-3. You can place **?** any place in a file name.

1-4. **ls [0-9]*** will list only those files that start with a number.

ls [a-m]* will list only those files that begin with letters "a" through "m".

1-5. If you placed the sequential command line in the background mode, the immediate system response was the PID for the job.

No, the **&** must be placed at the end of the command line.

1-6. The command line would be:

cd; pwd > junk; ls >> junk; ed trial<CR>

1-7. Change the **-c** option of the command line to read:

banner `date | cut −c1-10`<CR>

Shell Programming Exercises

2-1.

```
$cat time<CR>
banner `date | cut −c12-19`
$
$chmod u+x time<CR>
$ time<CR>
(banner display of the time 10:26)
$
```

2-2.

```
$cat mydate<CR>
banner `date | cut −c1-10`
$
```

2-3.

```
$cat tofriends<CR>
echo "Type in the name of the file containing the note."
read note
mail janice marylou bryan < $note
$
```

Or, if you wanted to use parameters for the logins.

```
$cat tofriends<CR>
echo "Type in the name of the file containing the note."
read note
mail $* < $note
$
```

2-4. **date | cut −c1-10 > file1<CR>**

2-5. **echo Dear colleague >> file1<CR>**

2-6.

```
$cat send.memo<CR>
date | cut −c1-10 > memo1
echo Dear colleague >> memo1
cat memo >> memo1
echo A memo from M. L. Kelly >> memo1
mail janice marylou bryan < memo1
$
```

2-7.

```
$cat mv.file<CR>
echo type in the directory path
read path
echo "type in file names, end with <^d>"
while
read file
    do
    for file
        in $file
        do
            mv $file $path/$file
        done
    done
echo all done
$
```

2-8.

```
$cat mv.file<CR>
echo Please type in directory path
read path
for file
    in $*
do
    mv $file $path/$file
done
$
```

The command line would then be:

```
$ mv.file *<CR>
$
```

2-9. See hint.

```
$ cat search<CR>
for file
  in $*
  do
    if grep $word $file >/dev/null
    then echo $word is in $file
    else echo $word is NOT in $file
    fi
  done
```

2-10. Type the following lines into your *.profile*.

```
stty -tabs<CR>
stty erase <^h><CR>
stty echoe<CR>
```

2-11. Type the following command line into your *.profile*

```
PS1="Hello<CR>" <CR>
```

2-12.

```
$ echo $HOME<CR>
```

```
$ echo $TERM<CR>
```

```
$ echo $PATH<CR>
```

Chapter 8

COMMUNICATION TUTORIAL

PAGE

INTRODUCTION 8-1

COMMUNICATING ON THE UNIX SYSTEM 8-2

HOW CAN YOU COMMUNICATE? 8-3

SENDING AND RECEIVING MESSAGES *(mail)* 8-4

- Sending Mail 8-4
  - Basics of Sending Mail 8-4
  - Sending Mail to One Person 8-5
  - Sending Mail to Several People Simultaneously 8-7
- Sending Mail to Remote Systems *(uname, uuname)* 8-8
- Receiving Mail 8-12

SENDING AND RECEIVING FILES 8-17

- Sending Small Files *(mail)* 8-17
- Sending Large Files *(uuto)* 8-19
  - Have You Got Permission? 8-19
- Sending a File *(uuto -m, uustat)* 8-21
- Receiving Files *(uupick)* 8-26

ADVANCED MESSAGE AND FILE HANDLING *(uucp, mailx)* 8-29

Chapter 8

COMMUNICATION TUTORIAL

INTRODUCTION

Sooner or later, you will want to use the UNIX system to get in touch with other UNIX system users. You may want to send a message to someone; the message may be one that must be read immediately. Perhaps you might need to send another user information from a file in your login.

Whatever the case, this chapter teaches you how to use the communication tools available to you on the UNIX system. The chapter begins with a brief overview of just who you might want to communicate with on the UNIX system. You learn how to send basic messages to users on your system and other UNIX systems, and also how to deal with messages you receive. You also learn about commands that enable you to send files to other users.

The following list is a review of the text conventions mentioned in *Chapter 2* that are used in this chapter.

bold	(Commands typed in exactly as shown.)
italic	(UNIX system prompts and responses.)
roman	(Input other than commands.)
<>	(Commands that are typed in, but are not reflected on the screen, are enclosed in angle brackets.)

COMMUNICATING ON THE UNIX SYSTEM

You can use the UNIX system to communicate with just about anyone else who uses the UNIX system. This means that your terminal does more than serve as a work station--it becomes your personal message-handling center as well, with the electronic equivalent of transmission, routing, and storage facilities.

Who would you want to communicate with over the UNIX system? Here are some examples to consider:

- The person in your office who needs to know about a department meeting tomorrow,
- Other users on your UNIX system who should see a posted message concerning their use of the system after office hours,
- The supervisor who wants a copy of your last two reports by 2:30 this afternoon,
- The supervisor who wants to review the memo you are presently working on as soon as you have finished it,
- A person working with you on the UNIX system to modify several files you both have in common; you need to be in touch from time to time, but the phones are being used as links from your terminals to the computer and you would rather not shout down the halls, and
- A coordinator who wants your daily operations records (all in very large files), but does not want to have to wade through them all at once when he receives them on the terminal.

As you can see, you can keep in touch with any number of people for any number of reasons through the UNIX system. The remainder of this chapter shows you how to use the various communication tools provided by the UNIX system to reach these people.

HOW CAN YOU COMMUNICATE?

The UNIX system offers several commands for user-to-user communication. This chapter explains the most important commands to know and suggests how to select the one to use in a given situation. The basic choice is between sending (or receiving) a message and sending (or receiving) a file.

To expand on one of the previous examples, suppose you are working at your terminal and you remember that you are giving a presentation at an officewide meeting tomorrow. You want to remind someone in your office about the presentation, but you do not want to take the time for a phone call or a walk to the other person's office. What can you do?

If the other person has a login on your UNIX system, you can use the **mail** command to send a brief message. When the recipient of your message finishes whatever task he or she is using the UNIX system for, a notice is posted that there is mail waiting to be read. The recipient can then read your message and send a reply back to your login.

To take another example, what if you need to send other people copies of things you already have on file--memos, reports, saved messages, documents, and the like? You can send such files using the **mail** command; however, this may not be the best way to send long files. For sending files over a page in length, you should use the **uuto** command. This command sends the file to a public directory on the recipient's system instead of sending it straight to the recipient's login. The recipient can then deal with it at his or her own leisure.

These are the important communication tools available to you. (Two other tools, the **uucp** and **mailx** commands, are discussed briefly at the end of the chapter.) Now that you have a general idea of how to communicate in the UNIX system, let's move on to the specifics.

SENDING AND RECEIVING MESSAGES *(mail)*

The **mail** command works in two ways--it lets you send messages to other UNIX system users, and it lets you read messages sent to you. This section deals first with sending messages, both to users on your UNIX system and to users on other UNIX systems that can communicate with yours.

Sending Mail

It is easy to send mail to another user. The basic command line format for sending mail is

mail login<CR>

where *login* is the recipient's login name on the UNIX system. This login name can be either of the following:

- A login name if the recipient is on your system, or
- A system name and login name if the recipient is on a system that can communicate with yours.

For the moment, assume that the recipient is on your system (known as the local system); we will deal with sending mail to users on other systems (known as remote systems) a little later.

Basics of Sending Mail

Since the recipient is on your system, you type the **mail** command as follows at the system prompt (**$**):

mail login<CR>
text

where *login* is the recipient's login name. Then you type in the text of the letter, as many lines as you need. When your message is complete, you send the message on its way by typing a dot (**.**) at the beginning of a new line.

The resulting message looks like this:

```
$ mail login<CR>
After you enter the command line,<CR>
type in as many lines of text as you need<CR>
to get the message across.<CR>
When you're done,<CR>
type in a control-d or a dot<CR>
on a line by itself, as shown on the next line.<CR>
.<CR>
$
```

The system prompt returns to notify you that your message has been queued (placed in line) and will be sent.

Sending Mail to One Person

Let's look at a sample situation. You have to notify another person in your office of a meeting later this afternoon, but he is not in and you have to leave your office. He has a login on your UNIX system with the login name *tommy*, so you can leave a message for him to read the next time he logs into the system:

```
$ mail tommy<CR>
Tom,<CR>
There's a meeting of the review committee<CR>
at 3:00 this afternoon.  D.F. wants your<CR>
comments and an idea of how long you think<CR>
the project will take to complete.<CR>
B.K.<CR>
.
$
```

When Tom logs in at his terminal (or while he is already logged in), he receives a message that tells him he has mail waiting:

```
you have mail
```

To see how *tommy* can read his mail, see the section titled *Receiving Mail*.

You can practice using the **mail** command by sending mail to yourself. This may sound strange at first, but it is the easiest way to practice sending messages. Simply type in the **mail** command and your own login name, then write a short message to yourself. When you type in the dot, the mail will be sent to your login and you will receive the notice that you have mail.

Sending mail to yourself can also serve as a handy reminder system. Suppose your login name is *rover;* you are ready to log off of the system for the day and you want to leave a reminder to call someone first thing the next morning. You might enter the following:

```
$ mail rover<CR>
Remember to call Accounting and find out<CR>
why they haven't returned my 1984 figures!<CR>
.
$
```

When you log in the next day, you will get a notice of messages awaiting you. Reading your mail then brings up the reminder message (and any other messages you may have received).

Sending Mail to Several People Simultaneously

If you need to send the same message to more than one person, simply place their login names after the **mail** command on the command line, with a space between each one, in the following format:

mail login1 login2 ... <CR>

where *login1, login2,* and ... are the different login names. You can mail messages to as many logins as you wish.

For example, if you send a notice about the department softball game to team members with login names *tommy, switch, wombat,* and *dave,* it might look like this:

```
$ mail tommy switch wombat dave<CR>
Diamond cutters,<CR>
The game is on for tonight at diamond three.<CR>
Don't forget your gloves!<CR>
Your Manager<CR>
.<CR>
$
```

To provide you with a quick summary of what you can expect when using the **mail** command to send messages, a recap of how to use it follows.

Command Recap

mail - sends a message to another user's login

command	*options*	*arguments*
mail	none	**login**

Description: **mail** followed by one or more login names, sends the message typed on the lines following the command line to the specified login(s).

Remarks: Typing a dot at the beginning of a new line sends the message.

Sending Mail to Remote Systems *(uname, uuname)*

We have assumed to this point that you are sending messages to recipients on your (local) UNIX system. You may have occasion, however, to send messages to recipients on other (remote) UNIX systems. For example, your office may have three separate systems, each in a different part of the building. Or perhaps you may have offices in several different locations, each with its own system.

How do you send mail to someone on a remote system? The UNIX system you are on must be able to communicate with a remote UNIX system before mail can be sent between the two. So, if you plan to send a mail message to someone on a remote system, you need to do a little legwork to find out the following information:

- Recipient's login name,
- Name of the remote system, and
- If your system and the remote system can communicate.

Two commands are available to help you answer these questions--the **uname** and **uuname** commands.

You can get the login name and the remote system name from the recipient. If it happens that the recipient does not remember the system name, have him or her log into the system and type the following at the system prompt:

uname -n<CR>

The **uname -n** command responds with the name of the system you are logged into. For example, if you are logged into a system named *sys10* and you type in **uname -n**, your screen should look like this:

```
$ uname -n<CR>
sys10
$
```

Once you know the remote system name, the **uuname** command helps you find out if your system can communicate with the remote system. At the prompt, type:

uuname<CR>

This generates a list containing the names of remote systems with which your system can communicate. If the recipient's system is in that list, then you can send messages there by **mail**.

The **uuname** command may respond with a large list of names if your system can communicate with many other systems. To avoid having that long list scroll quickly up your screen, use the pipe and **grep** command in conjunction with **uuname**. At the prompt, type:

uuname | grep system<CR>

where *system* is the recipient's system name. This generates the same list, then searches for and prints only the specified system name if it is found in the list.

For example, if you want to find out whether a system called *sys10* can communicate with your system, type:

```
$ uuname | grep sys10<CR>
```

If this is the case, the system name is printed in response:

```
$ uuname | grep sys10<CR>
sys10
$
```

If you get only the system prompt back, then the two systems cannot communicate:

```
$ uuname | grep sys10<CR>
$
```

Once you determine that you can send messages to a login on a remote system, your **mail** command line is slightly different than it is for sending mail to someone on your local system. The command line format for remote systems is:

mail system!login<CR>

where *system* is the remote system name and *login* is the recipient's login name. The two parts of the address are separated by an exclamation point (!).

Now that you have all the parts, let's put them together into an example. Assume that you have a message for someone on a different system in another part of your office. You know from the recipient her login name, *sarah*, and her system name, *sys10*. To find out if her system can communicate with yours, use the **uuname** command:

```
$ uuname | grep sys10<CR>
sys10
$
```

The system response tells you that your system is indeed networked to system *sys10*. Now all you have to do is send the message, using the expanded address format given previously:

```
$ mail sys10!sarah<CR>
Sarah,<CR>
The final counts for the writing seminar<CR>
are as follows:<CR>
<CR>
Our department - 18<CR>
Your department - 20<CR>
<CR>
Tom<CR>
.<CR>
$
```

Following is a quick summary of the two commands introduced in this section and what you can expect them to do.

Command Recap

uname - displays the system name

command	*options*	*arguments*
uname	**-n** and others*	none

Description: **uname -n** displays the name of the system on which your login resides.

* See the *UNIX System User Reference Manual* for all available options and an explanation of their capabilities.

Command Recap

uuname - displays a list of networked systems

command	*options*	*arguments*
uuname	none	none

Description: **uuname** displays a list of remote systems that can communicate with your system.

Receiving Mail

Once you learn to send messages, you may be anxious to read what others are sending your way. As stated earlier, the **mail** command also allows you to read messages sent by other UNIX system users.

After logging in, you may receive the following message at your terminal:

```
you have mail
```

This tells you that one or more messages are being held for you in a UNIX system directory named *usr/mail*, usually referred to as the the *mailbox*. Entering the **mail** command by itself allows you to read these messages.

To read your mail, type the **mail** command by itself at the system prompt:

mail<CR>

This displays the waiting messages at your terminal, one message at a time, with the most recently received message displayed first. In other words, as you read your messages, you go from the "newest" message to the "oldest" message.

A typical **mail** message looks like this:

```
$ mail
From tommy Mon May 21 15:33 CST 1984
B.K.
Looks like the meeting has been canceled.
Do you still want the technical review material?
Tom

?
```

The first line, called the *header,* displays information about a particular message--the login name of the sender, the date sent, and the time sent. The following lines (except for the last line) are the body of the message.

Notice the question mark (**?**) on the last line of the message. After displaying each message, the **mail** command displays a **?** and a space, and waits for a response from you before going on to the next message. There are several responses; we will look at the most common responses and what they do.

After reading a message, you may want to delete it. To do so, type a **d** after the question mark.

```
? d<CR>
```

This response deletes the message from the *mailbox* and displays the next message waiting in the *mailbox* (if there is one). If there are no other messages, the system prompt returns to indicate that you've finished reading your messages.

If you would rather display the next message without deleting the message being displayed, type a carriage return after the question mark.

```
? <CR>
```

The current message goes back into the *mailbox* and the next message is displayed. If there are no more messages in the *mailbox,* the system prompt returns.

You may want to save the message for later reference. To do so, type an **s** after the question mark:

```
? s<CR>
```

This response saves the mail message by default in a file called *mbox* in your login directory. If you would rather save the message in another file, follow the **s** response with a file name or with a path name ending in a file name.

For example, to save the message in a file called *mailsave* in your current directory, enter the following response after the question mark:

```
? s mailsave<CR>
```

If you use the **ls** command to list the contents of this directory, you will find the file *mailsave*.

You can also save the message in a file under another of your directories. If you have a mail message about a particular project or piece of work that you keep in a certain directory, you may want to save that message in the same directory. Let's say you have such a directory, named *project1*, under your login directory. If a mail

message comes in that you want to place in directory *project1*, under a file named *memo*, enter the following response after the question mark:

```
? s project1/memo<CR>
```

If you use the **cd** command to change directories from your login directory to *project1* and then use the **ls** command, you will find that the file *memo* is now listed. (You can use other, more complete path names as well; refer to *Chapter 3* for instruction on using path names.)

If you want to quit reading messages, enter the following response after the question mark:

```
? q<CR>
```

Any messages that you have left unread are put back in the *mailbox* until the next time you use the **mail** command.

If a long message is being displayed at your terminal, you can interrupt it by pressing the BREAK key. This stops the message display, prints the **?**, and waits for your response.

Other responses are available; these are listed in the *UNIX System User Reference Manual*. The following command recap summarizes what you can expect when using the **mail** command to read messages.

Command Recap

mail - reads messages sent to your login

command	*options*	*arguments*
mail	available*	none

Description: **mail** entered by itself displays any messages waiting in the system file *usr/mail* (the mailbox).

Remarks: The question mark (?) at the end of a message indicates that a response is expected. A full list of responses is given in the *UNIX System User Reference Manual.*

* See the *UNIX System User Reference Manual* for all available options and an explanation of their capabilities.

SENDING AND RECEIVING FILES

In several examples cited so far in this chapter, the need to send files from your UNIX system login to another UNIX system user has come up. Memos, reports, stories, baseball scores--there are numerous items that you can keep in your files. What do you do to send copies of those files to other UNIX system users?

Sending Small Files *(mail)*

The **mail** command uses the redirection symbol < to take its input from a specified file instead of from the keyboard. (For more detailed information on the use of redirection symbols, see *Chapter 7.*) The general format is as follows:

mail login < filename<CR>

where *login* is the recipient's login name and *filename* is the name of the file containing the information to be sent.

For example, assume you keep a standard meeting notice in a file named *meetnote*. If you want to send the letter to the owner of login *sarah* using the **mail** command, type the following at the prompt:

```
$ mail sarah < meetnote<CR>
$
```

The system prompt returns to let you know that the contents of *meetnote* have been sent. When *sarah* types in the **mail** command to read her messages, she will receive the standard meeting notice.

Likewise, if you want to send the same file to several users on your system, type in the **mail** command followed by the login names of the users, and then follow these with the < file redirection operator and the file name. It might look like this:

```
$ mail sarah tommy dingo wombat < meetnote<CR>
$
```

The system prompt tells you that the messages have been sent.

If the recipient for your file is on a remote system that can communicate with yours, simply redirect the file with the < operator:

mail system!login < filename<CR>

For example:

```
$ mail sys10!wombat < meetnote<CR>
$
```

Again, the system prompt notifies you that the message has been queued for sending.

Sending Large Files *(uuto)*

When you need to send large files, you should use the **uuto** command. This command can be used to send files to both local and remote systems. When the files arrive at their destination, the recipient receives a mail message announcing its arrival.

The basic format for the **uuto** command is

uuto filename system!login<CR>

where *filename* is the name of the file to be sent, *system* is the recipient's system, and *login* is the recipient's login name. The *filename* may be the name of a file or a path name ending in a specific file.

If you send a file to someone on your local system, you may omit the system name and use the following format:

uuto filename login<CR>

Have You Got Permission?

Before you actually send a file with the **uuto** command, you need to find out whether or not the file is transferable. To do that, you need to check the file's permissions. If they are not correct, you must use the **chmod** command to change them. (Permissions and the **chmod** command are covered in detail in *Chapter 3*.)

There are two permission criteria that must be met before a file can be transferred using **uuto**:

- The file to be transferred must have read permission (**r**) for *others,* and
- The directory that contains the file must have read (**r**) and execute (**x**) permission for *others.*

This may sound confusing, but an example should clarify the matter.

Assume that you have a file named *chicken,* under a directory named *soup,* that you want to send to another user with the **uuto** command. First you check the permissions on *soup,* which is under your login directory:

```
$ ls -l<CR>
total 35
-rwxr-xr-x   1  reader group1   5598   Mar 313:00  memos
drwxr--r--   2  reader group1    477   Mar 109:08  lists
drwxr-xr-x   2  reader group1     45   Feb 9 10:43 soup
$
```

Checking the line that contains the information for directory *soup* shows that it has read (**r**) and execute (**x**) permissions in all three groups; no changes have to be made. Now you use the **cd** command to change from your login directory to *soup* and then check the permissions on the file *chicken*:

```
$ ls -l chicken<CR>
total 4
-rw------- 1 reader group1 3101 Mar 1 18:22 chicken
$
```

The output informs you that the file *chicken* has read permission for you, but not for the rest of the system. To add those read permissions, you use the **chmod** command:

```
$ chmod go+r chicken<CR>
```

This adds read permissions to the rest of the system--*group* (**g**) and *others* (**o**)--without changing the previous permissions. Now, checking again with the **ls −l** command reveals the following:

```
$ ls −l chicken<CR>
-rw-r--r-- 1 reader group1 3101 Mar 1 18:22 chicken
$
```

This confirms that the file is now transferable using the **uuto** command. After you send copies of the file, you can reverse the procedure and replace the previous permissions.

Sending a File *(uuto −m, uustat)*

Now that you know how to determine if a file is transferable, let's take an example and see how the whole thing works.

The process of sending a file by **uuto** is referred to as a *job*. When you enter a **uuto** command, your job is not sent immediately. First it is stored in a queue (a waiting line of jobs) and assigned a job number. When the job's number comes up, it is transmitted to the remote system and placed in a public directory there. The recipient is notified by **mail** message and must use the **uupick** command to retrieve the file (this command is discussed later in the chapter).

For the following discussions, assume this information:

wombat	Your login name.
sys10	Your system name.
marie	Recipient's login name.
sys20	Recipient's system name.
money	File to be sent.

Also assume that the two systems can communicate with each other.

To send the file *money* to login *marie* on system *sys20,* enter the following:

```
$ uuto money sys20!marie<CR>
$
```

The system prompt returns, notifying you that the file has been sent to the job queue. The job is now out of your hands; all you can do is wait for confirmation that the job reached its destination.

How do you know when the job has been sent? The easiest method is to alter the **uuto** command line by adding a **−m** option, like so:

```
$ uuto −m money sys20!marie<CR>
$
```

This option sends a **mail** message back to you when the job has reached the recipient's system. The message may look something like this:

```
$ mail<CR>
From uucp Tue Apr 3 09:45 EST 1984
file /sys10/wombat/money, system sys10
copy succeeded

?
```

If you would rather check from time to time while you are working on the system, you can use the **uustat** command. This command keeps track of all the **uuto** jobs you submit and gives you their status. For example,

```
$ uustat<CR>
1145 wombat sys20 10/05-09:31 10/05-09:33 JOB IS QUEUED
$
```

The elements of this sample status message are as follows:

- *1145* is the job number associated with sending file *money* to *marie* on *sys20*.
- *wombat* is your login name.
- *sys20* is the recipient's system.
- *10/05-09:31* is the date and time the job was queued.

- *10/05-09:33* is the date and time of this particular **uustat** message.
- The final part is the status of the job--in this case indicating that the job has been queued, but has not yet been sent.

If you are interested in just one **uuto** job, you can use the **−j** option and the job number when requesting job status:

uustat −jjobnumber<CR>

In the example, let's say you enter the **uustat** command with the **−j** option (for job 1145) until you receive the following response:

```
$ uustat −j1145<CR>
1145 wombat sys20 10/05-09:31 10/05-09:37 COPYFINISHED,JOB DELETED
$
```

This status message indicates that the job was sent and has been deleted from the job queue--in other words, it has reached the public directory of the recipient's system. There are other status messages and options for the **uustat** command which are described in the *UNIX System User Reference Manual.*

That is all there is to sending files. You can practice simply by sending another UNIX system user a file. You should practice with a test file until you have the procedures down pat.

The following command recaps give a summary of the **uuto** and **uustat** commands for your convenience.

Command Recap

uuto - sends files to another login

command	*options*	*arguments*
uuto	**−m** and others*	**file system!login**

Description: **uuto** sends the specified file to the public directory of the specified system. The owner of the login is notified by **mail** that a file has arrived.

Remarks: Files to be sent must have read permission for *others;* the directory above the file must have read and execute permissions for *others.*

The **−m** option notifies you by mail when the file arrives at its destination.

* See the *UNIX System User Reference Manual* for all available options and an explanation of their capabilities.

Command Recap

uustat - checks job status of a **uuto** job

command	*options*	*arguments*
uustat	**-j** and others*	none

Description: **uustat** checks on the status of all **uuto** jobs sent from your login and displays the results.

Remarks: The **-j** option, followed by a specific job number, displays the status of only the specified job.

* See the *UNIX System User Reference Manual* for all available options and an explanation of their capabilities.

Receiving Files *(uupick)*

When a file sent by **uuto** shows up in the public directory on your UNIX system, you receive a **mail** message telling you that the file has arrived and where you can find it. To continue our previous example, let's see what the owner of login *marie* receives when she types in the **mail** command, not long after you (login *wombat*) have sent her the file *money*:

```
$ mail
From uucp Mon May 14 09:22 EST 1984
/usr/spool/uucppublic/receive/marie/sys10//money from sys10!wombat arrived
$
```

The message contains the following pieces of information:

- The first line tells you when the file arrived at its destination.
- The second line up to the two slashes (//) gives you the path name to the part of the public directory where the file has been stored.
- The second line after the two slashes tells you the name of the file and who sent it.

Once you have disposed of the **mail** message, you can use the **uupick** command to store the file where you want it. Type

uupick<CR>

at the system prompt. The command searches the public directory for any files sent to you. If it finds any, it prompts you with a **?** to do something with the file (much like the **mail** command).

Continuing with our previous example, if the owner of login *marie* enters the **uupick** command, she receives the following response:

```
$ uupick<CR>
from system sys10: file money
?
```

After the question mark (**?**), the command goes to the next line and waits for your response. There are several available responses; we will look at the most common responses and what they do.

The first thing you should do is move the file from the public directory and place it in your login directory so you can see what it is. To do so, type an **m** after the question mark.

```
?
m<CR>
$
```

This response moves the file into your current directory. If you wish to put it in some other directory instead, follow the **m** response with the directory name:

```
?
m directory<CR>
```

If there are other files waiting to be moved, the next one is displayed, followed by the question mark. If not, the prompt returns.

If you would rather display the next message without doing anything to the current file, press the carriage return key after the question mark.

```
?
<CR>
```

The current file remains in the public directory until you next use the **uupick** command. If there are no more messages, the system prompt returns.

If you already know that you do not want to save the file, you can delete it by typing in a **d** after the question mark:

```
?
d<CR>
```

This response deletes the current file from the public directory and displays the next message (if there is one). If there are no additional messages about waiting files, the prompt returns.

Finally, if you want to stop the **uupick** command, type a **q** after the question mark:

```
?
q<CR>
```

Any unmoved or undeleted files will wait in the public directory until the next time you use the **uupick** command.

Other available responses are listed in the *UNIX System User Reference Manual*. The following command recap summarizes what you can expect from the **uupick** command.

Command Recap

uupick - searches for files sent by **uuto**

command	*options*	*options*
uupick	none	none

Description: **uupick** searches the public directory of your system for files sent by **uuto**. If any are found, the command displays information about the file and awaits a response.

Remarks: The question mark (**?**) at the end of the message indicates that a response is expected. The full list of responses is given in the *UNIX System User Reference Manual*.

ADVANCED MESSAGE AND FILE HANDLING *(uucp, mailx)*

Once you master the **mail** and **uuto/uupick** commands, you may decide that you want commands that are more flexible or efficient. If so, you should try the **mailx** and **uucp** commands.

The **uucp** command enables you to send a copy of a file directly to another user's login directory, instead of to the public directory on that user's system. In some cases, you can even copy directly from

files in another login and place the copy in your login directory. The **uucp** command also enables you to rename a file when it reaches its destination.

There are a number of considerations to deal with when using **uucp**, such as file permissions and system security procedures. The **uucp** system is more complex and requires more experience to use than **uuto** and **uupick**.

If you want an electronic mail facility with more features, there is the **mailx** command. This command is an interactive message-handling system that gives you, among other things, the following:

- The ability to use either the **ed** or **vi** text editor for use on incoming *and* outgoing messages,
- A list of waiting messages from which the user can decide which messages to deal with and in what order,
- Several options for saving files, and
- Commands for replying to specific messages and sending copies to other users (both of incoming and outgoing messages).

As you might gather, these two commands are complex and are not recommended for the beginning user. Because of this, we do not cover the uses of **uucp** or **mailx** in this guide. However, these commands are mentioned here because they may be available in your UNIX system package and are useful commands to know about.

Once you are thoroughly familiar with the standard tools for user communication, you may want to experiment with the **uucp** and **mailx** commands. Refer to the *UNIX System User Reference Manual* for more information on using these commands.

SUPPLEMENTARY INFORMATION AND REFERENCE TOOLS

Contents

Appendix A. Selected UNIX System Documentation

Appendix B. File System Organization

Appendix C. Summary of UNIX System Commands

Appendix D. Quick Reference to ed Commands

Appendix E. Quick Reference to vi Commands

Appendix F. Summary of Shell Programming Ingredients

Glossary

Index

APPENDICES

PAGE

Appendix A: SELECTED UNIX SYSTEM DOCUMENTATION A-1
Document Descriptions A-1
Ordering Documents A-4

Appendix B: FILE SYSTEM ORGANIZATION B-1
UNIX System Directories B-3

Appendix C: SUMMARY OF UNIX SYSTEM COMMANDS C-1
Command Descriptions C-3

Appendix D: QUICK REFERENCE TO ed COMMANDS D-1
Commands in Alphabetical Order D-1
Commands by Topic D-5

Appendix E: QUICK REFERENCE TO vi COMMANDS E-1
Commands in Alphabetical Order E-1
Commands by Topic E-8

Appendix F: SUMMARY OF SHELL PROGRAMMING INGREDIENTS F-1
Shell Command Language F-1
Shell Programming Constructs F-3

Appendix A

SELECTED UNIX SYSTEM DOCUMENTATION

The *UNIX System User Guide* is a general introduction to the UNIX system. Several documents are available for follow-up study and for further reference. This appendix highlights documents to which you should refer next for detailed information on the topics presented in this guide. In addition, it provides brief instructions for obtaining these documents.

The documents selected for the appendix conform to the scope of the *UNIX System User Guide;* your needs may be different. For example, your documentation requirements will depend upon your use of the UNIX system, the special add-on packages available to you, and the computer on which you run the UNIX system. You may require more advanced or more detailed documentation on such things as support tools, system administration, or error messages. If so, refer to the *Documentation Directory* described in this appendix or contact your AT&T Technologies Account Representative.

DOCUMENT DESCRIPTIONS

Table A-1 summarizes the additional documentation by select code number (the reference number you must use when ordering any of the documents) and title. Following are brief descriptions of these documents.

UNIX System Synopsis (US-S02)
Briefly describes UNIX System V, Release 2.0, and some add-on software.

UNIX System Documentation Directory (307-006)
A conceptual outline of how to learn about the UNIX system through the existing documentation.

TABLE A-1

UNIX System Documentation Arranged by Select Code

Select Code	UNIX System Document Title
307-020	Documentation Directory
307-100	User Guide
307-103	Programming Guide
307-108	Support Tools Guide
307-109	User Reference Manual
307-110	User Reference Manual
307-113	Programmer Reference Manual
307-116	Programmer Reference Manual
307-123	Shell Commands and Programming
307-126	Editing Guide
307-130	User Quick Reference
307-137	Visual Editor Quick Reference
US-S02	Synopsis of the UNIX System

UNIX System Editing Guide (307-126)

Contains beginning and advanced information on the editing programs (including **ed** and **vi**) available with the UNIX system.

UNIX System Programmer Reference Manual

Gives detailed instructions for programmers using the UNIX system. This manual covers system calls, library functions and subroutines, file formats, and miscellaneous facilities for the programmer.

The UNIX system is portable to many different computers. Reference manuals are presently available for two types of computer systems; check with your system administrator

before ordering to determine which manual most closely serves the needs of your particular system:

- For the UNIX system running on an AT&T 3B20 computer, order 307-116, or

- For the UNIX system running on a DEC* VAX 11-780 or 11-750 computer, order 307-113.

UNIX System Programming Guide (307-103)
Describes the programming languages and language aids available on the UNIX system, including C, FORTRAN-77, and libraries.

UNIX System Shell Commands and Programming (307-123)
Gives detailed instructions for using the shell command language. This manual covers shell commands, input/output redirection, keyword parameters, and control flow.

UNIX System Support Tools Guide (307-108)
Describes the various software tools available to aid a UNIX system user/programmer.

UNIX System User Reference Manual
Gives complete instructions on all standard UNIX system commands, including proper format and all options.

The UNIX system is portable to many different computers. Reference manuals are presently available for two types of computer systems; check with your system administrator before ordering to determine which manual most closely suits the needs of your particular system:

- For the UNIX System running on an AT&T 3B20 computer, order 307-110, or

---

* Trademark of Digital Equipment Corporation

- For UNIX system running on a DEC VAX 11-780 or 11-750 computer, order 307-109.

UNIX System Quick Reference (307-130)
A pocket reference card with brief summaries of common UNIX system commands.

UNIX System Visual Editor Quick Reference (307-137)
A pocket reference card with brief summaries of the UNIX system screen editor (**vi**) commands and formatting information.

ORDERING DOCUMENTS

You may order UNIX system documents through your AT&T Technologies Account Representative or directly from AT&T Technologies Customer Information Center (CIC). This appendix does not cover pricing information, which is subject to change. For current and complete document availability and pricing information, contact the CIC Commercial Sales Representatives at one of the following numbers:

Continental U.S.	**1-800-432-6600 (toll free)**
Elsewhere	**1-317-265-3339**

When you have the current prices and are ready to order, refer to the following checklist.

- All orders must be prepaid and are subject to state and local taxes. If your organization is tax-exempt, provide a copy of your exemption certificate in lieu of the sales tax (except in Alaska, Arkansas, Delaware, Hawaii, Illinois, Maine, Montana, New Hampshire, Oregon, South Dakota, and Vermont).

- When ordering from the Customer Information Center, please include the following information:

  — Six-digit select code of each document,
  — Title of each document,
  — Quantity of each document, and
  — Ship-to address.

- Make checks payable to AT&T Technologies.

- Send your order to:

  AT&T Technologies
  CIC Commercial Sales
  Post Office Box 19901
  Indianapolis, Indiana 46219

- Orders shipped outside the continental United States must include freight charges. Freight is prepaid for orders shipped within the continental United States. Quotes expire within 30 days of the date of reply.

Appendix B

FILE SYSTEM ORGANIZATION

To make full use of the capabilities of the file system, you must understand its organization of files and directories. This appendix summarizes the standard system directories provided and maintained by the UNIX operating system.

The file structure of the UNIX system is a treelike structure (or *hierarchy*), with directories and files descending and branching out from a single directory. This directory is called the *root* and is designated by a slash (/). One path from the root leads to a directory that, in turn, leads to the directory that you find yourself in when you log in (your *home directory*). Under your home directory, you can establish your own hierarchy of directories and files for organizing information.

Other paths lead from the root to *system directories*. These directories are available to all users. The system directories described in this book are common to all UNIX system installations; they are provided and maintained by the operating system. In addition to this standard set of directories, you may have other system directories available to you. To obtain a complete listing of all the directories and files in the root directory on your UNIX system, type:

ls –l /<CR>

When you understand the organization of directories and files in the UNIX system, you will be able to use path names to move around in the structure and find out what other directories contain. For example, you can move to the directory **/bin** (which contains UNIX system executable files) by typing:

cd /bin<CR>

and then list its contents by inputting one of the following command lines:

ls<CR>	for a list of file and directory names
ls −l<CR>	for a detailed list of the contents

Or, you can use the **ls** command to view the contents of the **/bin** directory from any directory. Type:

ls /bin<CR>	for a short listing
ls −l /bin<CR>	for a detailed listing

You may use the same commands to look at the contents of other system directories, substituting the desired directory name for **/bin**.

Figure B-1 shows the root and the major system directories belonging to it. On the following pages are brief descriptions of each system directory.

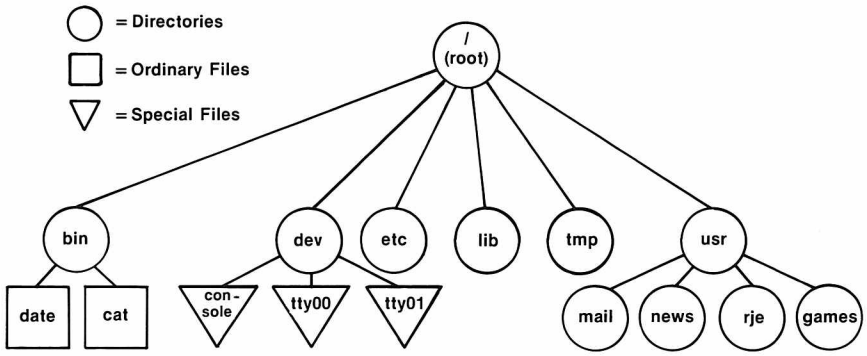


Figure B-1. Sample of file system structure of the UNIX system

UNIX SYSTEM DIRECTORIES

/ Root, the source of the file system.

/bin Many executable programs and utilities reside in this directory, such as:

cat
date
login
grep
mkdir
who

/lib This directory contains available program and language libraries, such as:

libc.a	system calls, standard I/O)
libm.a	math routines and support for languages such as C, FORTRAN, and BASIC.

/dev This directory contains special files that represent peripheral devices, such as:

console	console
lp	line printer
tty00	user terminal
tty01	user terminal
rp00	disks.

/etc Special programs and data files for system administration reside in this directory.

/tmp This directory holds temporary files, such as the buffers created for editing a file.

/usr This directory is the parent to the following subdirectories:

news	important news items
rje	data sent by the remote job entry
mail	electronic mail
games	electronic games
man	on-line user's manual
spool	files waiting to print on the line printer.

Appendix C

SUMMARY OF UNIX SYSTEM COMMANDS

This appendix consists of two sections.

- *Table C-1* summarizes the UNIX system commands covered in this guide. The table lists these commands alphabetically and supplies a brief definition for each one.
- The remainder of this appendix contains abridged descriptions of the capabilities of these commands.

TABLE C-1
Summary of UNIX System Commands

Name	Description
at	Specify time to run a job
banner	Display posters on the standard output
batch	Run jobs when system load permits
cat	Display contents of a file on the terminal
cd	Change your working directory
chmod	Change permission modes for a file or directory
cp	Copy an existing file to another file
cut	Cut out selected fields in a file
date	Display the current date and time

(Continued on next page)

TABLE C-1--*continued*

Name	Description
diff	Find difference(s) between two files
echo	Echo input to the standard output
ed	Edit (or create) a file using line editor
grep	Search a file for a pattern
kill	Terminate a background process
lex	Generate programs for simple lexical tasks
lp	Print a file on the line printer
lpstat	Display current line printer status
ls	List the contents of a directory
mail	Send or receive electronic mail
mailx	Interactive message handing system
make	Maintain large programs or documents
man	On-line manual
mkdir	Make (create) a new directory
mv	Move and rename a file
nohup	Continue background processes after logoff
pg	Display file contents a page at a time
pr	Display a partially formatted file on terminal
ps	Show status of background processes
pwd	Display the current working directory
rm	Remove (delete) a file
rmdir	Remove (delete) an empty directory
sh	Execute a shell file/program
sort	Sort or merge files

(Continued on next page)

TABLE C-1--*continued*

Name	Description
spell	Find spelling errors in a file
stty	Report or set I/O options for a terminal
uname	Print the name of the current UNIX system
uucp	Send a copy of a file directly to another user's login
uuname	List names of known remote systems
uupick	Retrieve a file in the public directory
uuto	Send a copy of a file to another user
uustat	**uuto** status inquiry
vi	Edit (create) a file using full screen editor
wc	Count lines, words, and characters in a file
who	Show who is logged into the system
yacc	Impose a structure on program input

COMMAND DESCRIPTIONS

at

Displays the job numbers of all jobs you have in the **at** or **batch** modes or in the background mode. Followed by a time, submits commands to be run at that time. A sample format for this command is:

```
at 0845am Jun 09<CR>
command1<CR>
command2<CR>
<^d>
```

If you use the **at** command without the date, the command executes within 24 hours at the time specified.

banner

Displays a message (in words up to 10 characters long) in large letters on the standard output.

batch

Submits command(s) to be processed when the system load is at an acceptable level. A sample format of this command is:

batch<CR>
command1<CR>
command2<CR>
<^d>

You may use a shell script for a command in **batch**. This may be useful and timesaving if you have a set of commands you frequently submit using the **batch** command.

cat

Displays the contents of a specified file at your terminal. To pause the output, use **<^s>**; **<^q>** resumes the display. To stop the display and return to the shell prompt, press the **<BREAK>** key.

cd

Changes your position in the file system from the current directory to your home directory. Followed by a directory name, this command changes your position in the file system from the current directory to the directory specified. You can move up or down in the file system. By using a path name in place of the directory name, you may jump several levels with one command.

cp

Copies a specified file into a new file. The **cp** command leaves the original file intact; if you do not want to retain the file as is, use **mv**.

cut

Cuts out specified fields from each line of a file. This command can be used to cut columns from a table, for example.

date

Displays the current date and time.

diff

Compares two files. The **diff** command reports which lines are different and what changes should be made to the second file to make it the same as the first file.

echo

Displays (echoes) input to the terminal on the standard output, followed by a carriage return.

ed

Edits a specified file using the line editor. If there is no file by that name, the **ed** command creates a file. See *Chapter 5* for detailed instructions on using the **ed** editor.

grep

Searches a specified file or files for a specified pattern and tells you which lines match. If you name more than one file, **grep** also tells you which file contains the pattern.

kill

Terminates a background process specified by its process identification number (**PID**). The **PID** can be found by using the **ps** command.

lex

Generates programs to be used in simple lexical analysis of text, perhaps as a first step in creating a compiler. See the *UNIX System User Reference Manual* for details.

lp

Prints a specified file on the line printer. This gives you a paper copy of the file's contents.

lpstat

Displays the status of any requests made to the line printer system. Options are available to request more detailed information.

ls

Lists the names of all files in the current directory except those whose names begin with a dot (**.**). Options are available to list more detailed information about the files in the directory.

mail

Displays any electronic mail you may have received at your terminal, one message at a time. Each message ends with **?**; type **r** for a list of options available to you at this point. There are options for saving, forwarding, or deleting mail.

When followed by a login name, **mail** sends a message to the user with the specified login name through electronic mail. You may type in as many lines of text as you wish; a dot (.) entered at the beginning of a new line ends the message and sends it to the recipient. Press the **<BREAK>** key to stop the mail session while composing the message or while reading one.

mailx

A more sophisticated, expanded version of electronic mail. See the *Unix System User Reference Manual* for details.

make

Provides a method for maintaining and supporting large programs or documents on the basis of smaller ones. See the *UNIX System User Reference Manual* for details.

man

Displays the manual page for a specified command at your terminal.

mkdir

Makes (creates) a new directory. The new directory becomes a subdirectory of the directory you are in when you issue the command. To create subdirectories or files in the new directory, you should move into the new directory with the **cd** command.

mv

Moves or renames a specified file. Either file name may be a path name. To make a copy of a file use the **cp** command.

nohup

Allows a command placed in the background to continue executing after you log off. Error messages are placed in a file called *nohup.out*.

pg

Displays the contents of a specified file at your terminal, a screenful at a time. After each screenful, the system pauses and waits for your instructions before proceeding.

pr

Displays a partially formatted version of a specified file at your terminal. The **pr** command shows page breaks, but does not implement any macros supplied for text formatter packages.

ps

Shows the status and number of all processes currently running. The **ps** command does not show the status of jobs in the **at** or **batch** queues, but it shows them when they are executing.

pwd

Displays the name of the current working directory. The **pwd** command shows the working directory with its full path name, beginning from the root. For an explanation of the file system organization, see *Appendix B*.

rm

Removes (deletes) a file. You may use metacharacters with the **rm** command, but with caution; a removed file cannot usually be recovered.

rmdir

Removes (deletes) a directory. The directory must be empty before you delete it; you must delete all files and subdirectories in the specified directory first.

sort

Sorts a file by the ASCII sequence and displays the results at your terminal. The sequence is as follows:

numbers before *letters*
capitals before *lower case*
alphabetical order

There are other options for sorting a file. For a complete list of **sort** options, see the *UNIX System User Reference Manual*.

spell

Collects words from a specified file and checks them against a spelling list. Words not on the list or not related to words on the list (with suffixes, prefixes, etc.) are displayed.

stty

By itself, reports the settings of certain input/output options for your terminal. It sets these options when followed with appropriate options and arguments (see the *UNIX System User Reference Manual*).

uname

Displays the name of the UNIX system on which your login resides.

uucp

Sends a specified file directly to a user's login. See the *Unix System User Reference Manual* for details.

uuname

Lists the names of remote UNIX systems that can communicate with your UNIX system.

uupick

Searches the public directory for files sent to you by the **uuto** command. If files are found, it displays the file name and the system it came from, then prompts you (**?**) to take action.

uustat

Displays the status of your request to send files to another user with the **uuto** command.

uuto

Sends a specified file to another user. The destination is in the format **system!login** where the **system** must be on the list of systems generated by the **uuname** command.

vi

Edits a specified file using the screen editor. If there is no file by that name, **vi** creates the file. See *Chapter 6* for detailed information on using the **vi** editor.

wc

Counts the number of lines, words, and characters in a specified file and displays the results at your terminal.

who

Displays the login names of the users logged in to the UNIX system on your computer, along with the terminals they are using and the times they logged in.

yacc

Imposes a structure on the input of a program. See the *UNIX System User Reference Manual* for details.

Appendix D

QUICK REFERENCE TO ed COMMANDS

This *Quick Reference to* **ed** *Commands* is organized into two sections.

- The first section lists the commands, with brief descriptions, in alphabetical order.
- The second section groups the commands according to each topic discussed in *Chapter 5*.

The commands are shown as you type them. The general format for **ed** commands is:

[address1,address2]command[parameter]<CR>

where *address1* and *address2* denote line addresses and *parameter(s)* indicates data on which the command operates. You can find complete information on using **ed** commands in *Chapter 5, Line Editor Tutorial.*

COMMANDS IN ALPHABETICAL ORDER

.	Returns to command mode from text input mode.
.	Address of the current line.
.	Matches any single character (in a search pattern and in a substitution).
!command	Temporarily escapes to the shell to execute a shell command.
/	Acts as a delimiter (for **s**, **v**, or **g** commands).
\	Removes the meaning of a special character (in a search pattern and in a substitution).

=	Displays the address of the last line in the buffer.
.=	Displays the current line number.
+*x*	Relative address; add *x* to the current line number.
−*x*	Relative address; subtract *x* from the current line number.
*	Matches zero or more occurrences of the preceding character (in search or substitution patterns).
.*	Matches zero or more occurrences of any characters following the period (in search or substitution patterns).
[...]	Matches the first character of those characters within the brackets.
[^...]	If the caret (^) is the first character in brackets, finds and matches the first character that is *not* within the brackets.
^	The caret (^) matches the beginning of a line (in a search pattern and in a substitution).
/pattern	Searches forward in the buffer and addresses the first line after the current line that contains the **pattern** of text.
?pattern	Searches backward in the buffer and addresses the first line before the current line that contains the **pattern** of text.
$	Denotes the last line in the buffer.
$	Matches the end of a line.
&	Repeats the last pattern to be substituted.

%	Repeats the last replacement pattern.
@	Deletes the current line (text input mode) or a command line (command mode).
#	Deletes the character just entered (text input mode).
a	Creates text after the specified line.
c	Replaces text in the specified lines with new text.
CR	Carriage return; moves down a line in the buffer.
d	Deletes specified lines of text.
ed filename	Enters **ed** line editor to edit a file called *filename;* copies the file into the buffer.
E filename	Replaces the current buffer with the contents of a file called *filename;* deletes present contents of the buffer whether written to a permanent file or not.
f	Displays the name of the file being edited.
f newfile	Changes the current file name associated with the buffer to *newfile.*
g/pattern	Addresses all lines in the buffer that contain the specified **pattern** of text.
G/pattern	Addresses all lines in the buffer that contain the specified **pattern** of text; prints each occurrence for you to deal with separately.
h	Displays a short explanation of the previous diagnostic response (**?**).
H	Automatically displays explanations of diagnostic responses (**?**) throughout the editing session.

i Inserts new text before the specified line.

j Joins contiguous lines.

l Displays the specified lines with all nonprinting (hidden) characters.

m Moves the specified lines after a destination line; deletes the lines at the old location.

n Displays the specified lines preceded by the line addresses and a tab space.

p Displays the specified lines in the buffer.

P Causes **ed** to print an asterisk (*) as a command mode prompt (for more details, see the *UNIX System Editing Guide* described in *Appendix A*).

q Ends an editing session. If changes to the buffer were not written to a file, a warning (**?**) is given. Typing **q** a second time ends the session without writing to a file.

Q Ends an editing session whether or not changes to the buffer were written into a file.

r filename Appends the contents of a file called *filename* to the end of the present buffer contents.

s/old text/new text/
Substitutes the first occurrence of **old text** with **new text** on the current line. A **g** after the final slash changes all occurrences on the current line.

address1,address2s/old text/new text/
Substitutes the first occurrence of **old text** with **new text** on the lines denoted by *address1,address2*.

t Copies the specified lines and places them after a destination line.

u	Undoes the last command given, except for **w** and **q** (command mode).
v/pattern	Addresses all lines in the buffer that *do not* contain the specified **pattern** of text.
V/pattern	Addresses all lines in the buffer that *do not* contain the specified **pattern** of text; prints each occurrence for you to deal with separately.
w	Copies the buffer contents into the file currently associated with the buffer.
w filename	Copies the buffer contents into a file called *filename*.

COMMANDS BY TOPIC

Commands for Getting Started

ed filename	Enters **ed** line editor to edit a file called *filename*.
a	Appends text after the current line.
.	Ends the text input mode and returns to the command mode.
p	Displays the current line.
d	Deletes the current line.
CR	Moves down one line in the buffer.
–	Moves up one line in the buffer.
w	Writes the buffer contents to the file currently associated with the buffer.

q	Ends an editing session. If changes to the buffer were not written to a file, a warning (**?**) is issued. Typing **q** a second time ends the session without writing to a file.

Line Addressing Commands

1, 2, 3...	Denotes line addresses in the buffer.
.	Address of the current line in the buffer.
.=	Displays the current line address.
$	Denotes the last line in the buffer.
,	Addresses lines 1 through the last line.
;	Addresses the current line through the last line.
$+x$	Relative address; add x to the current line number.
$-x$	Relative address; subtract x from the current line number.
/abc	Searches forward in the buffer and addresses the first line after the current line that contains the pattern *abc*.
?abc	Searches backward in the buffer and addresses the first line before the current line that contains the pattern *abc*.
g/abc	Addresses all lines in the buffer that contain the pattern *abc*.
v/abc	Addresses all lines in the buffer that *do not* contain the pattern *abc*.

Display Commands

p	Displays the specified lines in the buffer.
n	Displays the specified lines preceded by the line addresses and a tab space.

Text Input

a	Enters text after the specified line in the buffer.
i	Enters text before the specified line in the buffer.
c	Replaces text in the specified lines with new text.
.	On a line by itself, ends the text input mode and returns to the command mode.

Deleting Text

d	Deletes one or more lines of text (command mode).
u	Undoes the last command given (command mode).
@	Deletes the current line (text input mode) or a command line (command mode).
# or **backspace**	Deletes the last character typed in (text input mode).

Substituting Text

address1,address2s/old text/new text/command

Substitutes **new text** for **old text** within the range of lines denoted by *address1,address2* (which may be numbers, symbols, or text). The **command** may be **g**, **l**, **n**, **p**, or **gp**.

Special Characters

.	Matches any single character in search or substitution patterns.
*	Matches zero or more occurrences of the preceding character in search or substitution patterns.
.*	Matches zero or more occurrences of any characters following the period in search or substitution patterns.
^	The caret (^) matches the beginning of the line in search or substitution patterns.
$	Matches the end of the line in search or substitution patterns.
\	Takes away the special meaning of the special character that follows in search and substitution patterns.
&	Repeats the last pattern to be substituted.
%	Repeats the last replacement pattern.
[...]	Matches the first occurrence of a pattern in the brackets.
[^...]	Matches the first occurrence of a character that is *not* in the brackets.

Text Movement Commands

m	Moves the specified lines of text after a destination line; deletes the lines at the old location.
t	Copies the specified lines of text and places the copied lines after a destination line.
j	Joins the current line with the next contiguous line.
w	Copies (writes) the buffer contents into a file.
r	Reads in text from another file and appends it to the buffer.

Other Useful Commands and Information

h	Displays a short explanation for the preceding diagnostic response (**?**).
H	Turns on the help mode, which automatically displays an explanation for each diagnostic response (**?**) during the editing session.
l	Displays nonprinting (hidden) characters in the text.
f	Displays the current file name.
f newfile	Changes the current file name associated with the buffer to *newfile*.
!command	Temporarily escapes to the shell to execute a shell command.
ed.hup	If the terminal is hung up before a **write** command, the editing buffer is saved in the file *ed.hup*.

Appendix E

QUICK REFERENCE TO vi COMMANDS

This *Quick Reference to* **vi** *Commands* is organized into two sections.

- The first section lists the commands, with brief descriptions, in alphabetical order.
- The second section lists the commands according to each topic discussed in *Chapter 6*.

Please note the following conventions when using this appendix.

- Typing the control key is denoted by a caret (^) and the key--for example, **^g**.
- "Current line," "current word," and "current character" refer to the line, word, or character denoted by the cursor.

The commands are shown as you type them. The general format for **vi** commands is:

[x]command[argument]

where **x** denotes a number and *argument* indicates data on which the command operates. You can find complete information on using **vi** commands in *Chapter 6, Screen Editor Tutorial.*

COMMANDS IN ALPHABETICAL ORDER

;	Continues the search for the character specified by the **f** command.
.	Repeats the action initiated by the last command.
\	Prints the characters that are normally nonprinting (hidden characters) text input mode.
:	Begins a line editor command.

:$	Moves the cursor to the beginning of the last line in the buffer.
:*n*	Moves the cursor to the beginning of the *n*th line of the buffer (*n* = line number).
–	Moves the cursor up one line.
+	Moves the cursor down one line.
(	Moves the cursor to the beginning of the current sentence.
)	Moves the cursor to the beginning of the next sentence.
{	Moves the cursor to the beginning of the current paragraph.
}	Moves the cursor to the beginning of the next paragraph.
~	Change uppercase to lowercase or lowercase to uppercase.
/pattern	Searches forward in the buffer for **pattern**.
?pattern	Searches backward in the buffer for **pattern**.
@	In text input mode, erases the current line.
CR	Carriage return; in command mode, moves the cursor down one line.
space bar	Moves the cursor to the right one character.
a	Enters text after the cursor.
A	Enters text at the end of the current line.
b	Moves the cursor to the left one word, to the first character of that word.

B	Moves the cursor to the left one word (delimited only by blanks), to the first character of that word.
^b	Scrolls the screen back a full window, revealing the window of text above the current window.
BS	Backspace; in command mode, moves the cursor one character to the left.
BS	Backspace; in text input mode, deletes the current character.
cw	Replaces a word (or part of a word), from the cursor to the next space or punctuation, with new text.
cc	Replaces all the characters in the current line.
*n***c***x*	Replaces *n* number of text objects *x*, where *x* can include a sentence or a paragraph.
C	Replaces the characters in the current line from the cursor to the end of the line.
D	Deletes the line from the cursor to the end of the line.
^d	Scrolls the screen down a half window, revealing lines below the current window.
^d	Escapes the temporary return to the shell and returns to **vi** to edit the current window.
dd	Deletes the current line.
dw	Deletes a word (or part of a word) from the cursor through the next space or to the next punctuation.
*n***d***x*	Deletes *n* number of text objects *x*, where *x* can include a sentence or a paragraph.
:.,$d	Deletes all the lines from the current line to the last line.

e Moves the cursor to the end of the current word.

E Moves the cursor to the end of the current word (delimited by blanks only); places the cursor on the last character before the next blank space or at the end of the current line.

ESC Escape; returns to the command mode from any of the text input modes.

f*x* Moves the cursor right to the specified character *x*.

F*x* Moves the cursor left to the specified character *x*.

^f Scrolls the screen forward a full window, revealing the window of text below the current window.

G Moves the cursor to the beginning of the last line in the buffer.

***n*G** Moves the cursor to the *n*th line of the file (*n* = line number).

^g Gives the line number, its position in the buffer (as a percentage of the buffer completed), and status.

:g/text/s//new words/g
Changes every occurrence of **text** to **new words**.

h Moves the cursor one character to the left.

H Moves the cursor to the first line on the screen, or "home".

i Enters text to the right of the cursor.

I Enters text to the left of the first character that is not a blank on the current line.

j Moves the cursor down one line from its present position.

J Joins the line immediately below the current line with the current line.

k Moves the cursor up one line from its present position.

l Moves the cursor one character to the right.

L Moves the cursor to the last line on the screen.

^l Clears and redraws the current window.

M Moves the cursor to the middle line on the screen.

n Repeats the last search command.

N Repeats the last search command in the opposite direction.

o Enters text at the beginning of a new line immediately below the current line.

O Enters text at the beginning of a new line immediately above the current line.

p Places the contents of the temporary buffer containing the last "delete" or "yank" command into the text after the cursor or below the current line.

"*x*p Places the contents of register *x* after the cursor or below the current line.

:q Quits **vi** if changes made to the buffer were written to a file.

:q! Quits **vi** whether or not changes made to the buffer were written to a file.

r Replaces the current character.

R Replaces only those characters that are typed over with new text; continues to append new text after the end of the line is reached.

:r filename
Inserts the contents of a file called *filename* under the current line of the buffer.

s Deletes the current character and appends text.

S Replaces all the characters in the current line.

:s/text/new words/
Replaces **text** with **new words** on the current line.

:s/text/new words/g
Changes every occurrence of **text** on the current line to **new words**.

t*x* Moves the cursor right to the character just before the specified character *x*.

T*x* Moves the cursor left to the character just after the specified character *x*.

u Undoes the last command.

U Erases the last change on the current line.

^u Scrolls the screen up a half window, revealing the lines of text above the current window.

^v In text input mode, prints characters that are normally nonprinting (hidden characters).

vi filename
Enters **vi** screen editor to edit the file *filename*.

vi file1 file2 file3
Enters three files into the **vi** buffer to be edited. Those files are *file1*, *file2*, and *file3*.

vi −r file1
Restores the changes made to file *file1* that were lost because of an interrupt in the system.

view file1
Views file *file1* in the read-only mode of **vi**.

w Moves the cursor forward to the first character in the next word.

W Ignores all punctuation and moves the cursor forward to the beginning of the next word delimited only by blanks.

:w filename
:n When editing more than one file, writes the buffer to the file called *filename* and calls the next file in the buffer (use **:n** only after **:w**).

:w filename
:q Writes the buffer to a new file called *filename* and quits **vi**.

:wq Writes the buffer to a file and quits **vi**.

:*x*,*z*w data
Writes lines from the number *x* through the number *z* into a new file called **data**.

:w! filename
:q Overwrites an existing file called *filename* and quits **vi**.

^w In text input mode, deletes the current word delimited by blanks.

x Deletes the current character.

*n*y*x* Places (yanks) a copy of *n* numbertext objects *x* into a temporary buffer, where *x* can include a word, line, sentence, or paragraph.

"*l*y*n* Places a copy of text object *x* into a register named by a letter *l*.

yy Places the current line of text into a temporary buffer.

ZZ Writes the buffer to a file and quits **vi**

SUMMARY OF vi COMMANDS BY TOPIC

Commands for Getting Started

TERM=code

export TERM
Before entering **vi**, sets the terminal configuration.

vi filename
Enters **vi** screen editor to edit a file called *filename*.

a	Enters text after the cursor.
h	Moves the cursor to the left one character.
j	Moves the cursor down one line.
k	Moves the cursor up one line.
l	Moves the cursor to the right one character.
x	Deletes the current character.
CR	Carriage return; moves the cursor down to the beginning of the next line.
ESC	Escape; leaves text input mode and returns to command mode.
ZZ	Writes to a file and quits **vi**.
:q	Quits **vi** if changes made to the buffer were written to a file.

Commands for Positioning in the Window

Character Positioning

h	Moves the cursor one character to the left.
l	Moves the cursor one character to the right.

BS	Backspace; moves the cursor one character to the left.
space bar	Moves the cursor one character to the right.
f*x*	Moves the cursor right to the specified character *x*.
F*x*	Moves the cursor left to the specified character *x*.
;	Continues the search for the character specified by the **f** command. It will remember the character and seek out the next occurrence of the character on the current line.
t*x*	Moves the cursor right to the character just before the specified character *x*.
T*x*	Moves the cursor left to the character just after the specified character *x*.

Positioning by Lines

j	Moves the cursor down one line from its present position.
k	Moves the cursor up one line from its present position.
+	Moves the cursor down one line.
–	Moves the cursor up one line.
CR	Carriage return; moves the cursor down to the beginning of the next line.

Word Positioning

w	Moves the cursor to the right, to the first character in the next word.
W	Ignores all punctuation and moves the cursor to the right, to the beginning of the next word delimited only by blanks.

b Moves the cursor to the left one word, to the first character of that word.

B Moves the cursor to the left one word, (delimited only by blanks) to the first character of that word.

e Moves the cursor to the end of the current word.

E Moves the cursor to the end of the current word (delimited by blanks only); places the cursor on the last character before the next blank space or at the end of the current line.

Positioning by Sentences

(Moves the cursor to the beginning of the current sentence.

) Moves the cursor to the beginning of the next sentence.

Positioning by Paragraphs

{ Moves the cursor to the beginning of the current paragraph.

} Moves the cursor to the beginning of the next paragraph.

Positioning in the Window

H Moves the cursor to the first line on the screen, or "home".

M Moves the cursor to the middle line on the screen.

L Moves the cursor to the last line on the screen.

Commands for Positioning in the File

Scrolling

^f	Scrolls the screen forward a full window, revealing the window of text below the current window.
^d	Scrolls the screen down a half window, revealing lines of text below the current window.
^b	Scrolls the screen back a full window, revealing the window of text above the current window.
^u	Scrolls the screen up a half window, revealing the lines of text above the current window.

Positioning on a Numbered Line

G	Moves the cursor to the beginning of the last line in the buffer.
^g	Gives the line number, position in the buffer (as a percentage of the buffer completed), and status.
*n***G**	Moves the cursor to the *n*th line of the file (*n* = line number).

Searching for a Pattern

/pattern	Searchs forward in the buffer for the next occurrence of the **pattern** of text. Positions the cursor under the first character of the pattern.
?pattern	Searchs backward in the buffer for the first occurrence of **pattern** of text. Positions the cursor under the first character of the pattern.
n	Repeats the last search command.
N	Repeats the search command in the opposite direction.

Create Commands

a	Enters text after the cursor.
A	Enters text at the end of the current line.
i	Enters text to the right of the cursor.
I	Enters text to the right the first character that is not a blank on the current line.
o	Enters text at the beginning of a new line immediately below the current line.
O	Enters text at the beginning of a new line immediately above the current line.
ESC	Escape; returns to the command mode from any of the above text input modes.

Delete Commands

For the TEXT INPUT Mode

BS	Backspace; deletes the current character.
^w	Deletes the current word delimited by blanks.
@	Erases the current line of text.

For the COMMAND Mode

u	Undoes the last command.
U	Erases the last change on the current line.
x	Deletes the current character.
dw	Deletes a word (or part of a word) from the cursor through the next space or to the next punctuation.

dd	Deletes the current line.
*n***d***x*	Deletes *n* number of text objects *x*. *x* can be the symbol for a word, line, current sentence, or current paragraph.
D	Deletes the current line from the cursor to the end of the line.

Change Commands

r	Replaces the current character.
R	Replaces only those characters typed over with new characters; continues to append new text after the end of the line until the **ESC** command is given.
s	Deletes the current character and appends text until the **ESC** command is given.
S	Replaces all the characters in the current line.
cw	Replaces the current word or the remaining characters in the current word with new text, from the cursor to the next space or punctuation.
cc	Replaces all the characters in the current line.
*n***c***x*	Replaces *n* number of text objects *x*. *x* can be the symbol for a word, line, current sentence, or current paragraph.
C	Replaces the remaining characters in the current line, from the cursor to the end of the line.

Cut and Paste Commands

p	Places the contents of the temporary buffer containing the last "delete" or "yank" command into the text after the cursor or below the current line.
yy	Places (yanks) a line of text into a temporary buffer.
*n***yx**	Places a copy of *n* number of text objects *x* into a temporary bufffer.
"*l***y***x*	Places a copy of text object *x* into a register named by a letter *l*. *x* can be the symbol for a word, line, current sentence, or current paragraph.
"*x***p**	Places the contents of register *x* after the cursor or below the current line.

Special Commands

.	Repeats the action initiated by the last command.
J	Joins the line immediately below the current line with the current line.
****	Prints characters that are normally nonprinting (hidden characters) in text input mode.
^v	Prints characters that are normally nonprinting (hidden characters) in text input mode.
^l	Clears and redraws the current window.
~	Change uppercase to lowercase or lowercase to uppercase.

Line Editor Commands

: Tells **vi** that the next commands are line editor commands.

:sh Temporarily returns to the shell to perform some shell commands without leaving **vi**.

^d Escapes the temporary return to the shell and returns to **vi** to edit the current window.

:*n* Goes to the *n*th line of the buffer.

:*x*,*z***w data**
Writes lines from the number *x* through the number *z* into a new file called **data**.

:$ Moves the cursor to the beginning of the last line in the buffer.

:.,$d Deletes all the lines from the current line to the last line.

:r filename
Inserts the contents of the file *filename* under the current line of the buffer.

:s/text/new words/
Replaces the first instance of **text** on the current line with **new words**.

:s/text/new words/g
Replace every occurrence of **text** on the current line with **new words**.

:g/text/s//new words/g
Changes every occurrence of **text** in the buffer to **new words**.

Quit Commands

ZZ Writes the buffer to a file and quits **vi**.

:wq Writes the buffer to a file and quits **vi**.

:w filename
:q
Writes the buffer to a new file named *filename* and quits **vi**.

:w! filename
:q Overwrites an existing file called *filename* with the contents of the buffer and quits **vi**.

:q! Quits **vi** whether or not changes made to the buffer were written to a file.

:q Quits **vi** if changes made to the buffer were written to a file.

Special Options for vi

vi file1 file2 file3
Enters three files into the **vi** buffer to be edited. Those files are *file1*, *file2*, and *file3*.

:w
:n When editing more than one file, writes the buffer to a file called *filename* and calls the next file in the buffer (use **:n** only after **:w**).

vi −r file1
Restores the changes made to *file1* that were lost because of an interrupt in the system.

view file1
Views *file1* in the read-only mode of **vi**. Changes cannot be made to the buffer.

Appendix F

SUMMARY OF SHELL PROGRAMMING INGREDIENTS

This summary of shell programming ingredients discussed in *Chapter 7, Shell Tutorial,* is organized into two sections.

- The first section is a summary of the variables and special symbols of the shell. These are arranged by topic in the order that they were discussed in the chapter.
- The second section shows the shell programming constructs.

SHELL COMMAND LANGUAGE

Special Characters in the Shell

* ? []"	Metacharacters; used as file name shortcuts (file name generation).
&	Executes commands in the background mode.
;	Sequentially executes several commands typed in on one line, each separated by ;.
\	Turns off the meaning of special characters in the shell.
'...' "..."	Single quotes turn off the special meaning of all characters. Double quotes allow $, `, and " to retain their special meaning.

Redirecting Input and Output

<	Redirects the contents of a file into a command.

>	Redirects the output of a command into a new file, or replaces the contents of an existing file with the output.
>>	Redirects the output to be added to the end of a file.
|	Directs the output of one command to be the input of the next command.
`command`	Substitutes the output of the enclosed command line.

Executing and Terminating Processes

batch	Submits the commands that follow to be processed at a time when the system load is at an acceptable level. **^d** ends the **batch** command.
at	Submits the following commands to be executed at a specified time. **^d** ends the **at** command.
at −l	Gives the current jobs in the **at** or **batch** queue.
at −r	Removes the **at** or **batch** job from the queue.
ps	Gives the status of the shell processes.
kill PID	Terminates the shell process with the specified process ID (PID).
nohup command list &	Completes background processes after logging off.

Executing A File

sh filename	Executes a shell file that is a program.
chmod u+x filename	Changes the mode of a file to be executable by you.
bin	Your directory for storing executable shell programs that are accessible to all of your other directories.

Variables

positional parameter	A variable defined by its position on the command line.

$# Gives the number of positional parameters.

$* Substitutes all positional parameters starting with the first positional parameter.

named variable
A variable that is given a name by you.

Variables Used by the Shell

HOME	Denotes your home directory; the default variable for the **cd** command.
PATH	Defines the path your login shell follows to find commands.
CDPATH	Defines the search path for the **cd** command.
MAIL	Gives the name of the file containing your electronic mail.
PS1 PS2	Defines the primary and secondary prompt strings.
TERM	Defines the type of terminal.
IFS	Defines the internal field separators; normally the space, the tab, and the carriage return.

SHELL PROGRAMMING CONSTRUCTS

Here Document

```
command <<!
input lines
!
```

For Loop

```
for variable<CR>
        in this list of values<CR>
do the following commands<CR>
        command 1<CR>
        command 2<CR>
            .<CR>
            .<CR>
        last command<CR>
done<CR>
```

While Loop

```
while command list<CR>
do<CR>
    command1<CR>
    command2<CR>
        .<CR>
        .<CR>
    last command<CR>
done<CR>
```

If...Then

```
if this command is successful<CR>
then command1<CR>
    command2<CR>
        .<CR>
        .<CR>
    last command<CR>
fi<CR>
```

If...Then...Else

```
if command list<CR>
    then command list<CR>
    else command list<CR>
fi<CR>
```

Case Construction

```
case characters<CR>
in<CR>
    pattern1)<CR>
        command line 1<CR>
            .<CR>
            .<CR>
        last command line<CR>
    ;;<CR>
    pattern2)<CR>
        command line 1<CR>
            .<CR>
            .<CR>
        last command line<CR>
    ;;<CR>
    pattern3)<CR>
        command line 1<CR>
            .<CR>
            .<CR>
        last command line<CR>
    ;;<CR>
esac<CR>
```

break Statement

This statement forces the program to leave any loop and execute the next command.

GLOSSARY

This glossary defines terms and acronyms used in the *UNIX System User Guide* that may not be familiar to you.

acoustic coupler
A device that permits transmission of data over an ordinary telephone line. When you place a telephone handset in the coupler, you link a computer at one end of the phone line to a peripheral device, such as a user terminal, at the other.

address
Generally, a number that indicates the location of information in the computer's memory. In the UNIX system, the address is part of an editor command that specifies a line number or range.

append mode
A text editing mode where you enter (append) text after the current position in the buffer. See **text input mode**, compare with **command mode** and **insert mode**.

argument
Special instructions on the command line that specify data on which a command is to operate. Arguments usually follow the command and can include numbers, letters, or text strings. For instance, in the command **lp −m myfile**, **lp** is the command and **myfile** is the argument. See **option**.

ASCII
(pronounced **as'-kee**) American Standard Code for Information Interchange, a standard for data transmission that is used in the UNIX system. ASCII assigns sets of 0s and 1s to represent 128 characters, including alphabetical characters, numerals, and standard special characters, such as #, $, %, and &.

AT&T 3B computers
Computers manufactured by AT&T Technologies, Inc.

background

A type of program execution where you request the shell to run a command away from the interaction between you and the computer ("in the background"). While this command runs, the shell prompts you to enter other commands through the terminal.

baud rate

A measure of the speed of data transfer from a computer to a peripheral device (such as a terminal) or from one device to another. Common baud rates are 300, 1200, 4800, and 9600. As a general guide, divide a baud rate by 10 to get the approximate number of English characters transmitted each second.

buffer

A temporary storage area of the computer used by text editors to make changes to a copy of an existing file. When you edit a file, its contents are read into a buffer, where you make changes to the text. For the changes to become a part of the permanent file, you must write the buffer contents back into the file. See **permanent file**.

child directory

See **subdirectory**.

command

The name of a file that contains a program that can be processed or executed by the computer on request.

command file

See **executable file**.

command language interpreter

A program that acts as a direct interface between you and the computer. In the UNIX system, a program called the **shell** takes commands and translates them into a language understood by the computer.

command line

A line containing one or more commands, ended by typing a carriage return (**<CR>**). The line may also contain options and arguments. You type this line to the shell to instruct the computer to perform one or more tasks.

command mode

A text editing mode in which each character you type is interpreted as an editing command. This mode permits actions such as moving around in the buffer, deleting text, or moving lines of text. See **text input mode**, compare with **append mode** and **insert mode**.

context search

A technique for locating a specified pattern of characters (called a string) when in a text editor. Editing commands that cause a context search scan the buffer, looking for a match with the string specified in the command. See **string**.

control character

A nonprinting character that is entered by holding down the control key and typing a character. A control character transmits a special command to the computer. For instance, when viewing a long file on your screen with the **cat** command, typing control-s (**^s**) stops the display so you can read it, and typing control-q (**^q**) continues the display.

current directory

The directory in which you are presently working. You have direct access to all files and subdirectories contained in your current directory. The shorthand notation for the current directory is a dot (.).

cursor

A cue printed on the terminal screen that indicates the position at which you enter or delete a character. It is usually a rectangle or a blinking line.

default

An automatically assigned value or condition that exists unless you explicitly change it. For example, the shell prompt string has a default value of **$** unless you change it.

delimiter

A character that logically separates items or arguments on a command line. Two frequently used delimiters in the UNIX system are the space and the tab. Another is the slash character (/) that separates directories from subdirectories and files in a path name.

diagnostic

A message printed at your terminal to indicate an error encountered while trying to execute some command or program. Generally, you need not respond directly to a diagnostic message.

directory

A type of file used to group and organize other files or directories. You cannot enter text or other data into a directory. (For more detail, see *Appendix B, File System Organization.*)

disk

A magnetic data storage device consisting of several round plates similar to phonograph records. Disks store large amounts of data and allow quick access to any piece of data.

electronic mail

The feature of an operating system that allows computers users to exchange written messages via the computer. The UNIX system **mail** command provides electronic mail in which the addresses are the login names of users.

environment

The conditions under which you work while using the UNIX system. Your environment includes those things that personalize your login and allow you to interact in specific ways with the UNIX system and the computer. For example, your shell environment includes such things as your shell prompt string, specifics for backspace and erase characters, and commands for sending output from your terminal to the computer.

erase character

The character you type to delete the previous character on the current line. The UNIX system default erase character is #.

escape

A means of getting into the shell from within a text editor or another program.

execute

The computer's action of interpreting a programmed instruction or command and performing the indicated operation(s).

executable file

A file that can be processed or executed by the computer without any further translation. When you type in the file name, the commands in the file are executed. See **shell procedure**.

field

A word or a group of characters treated as one word on a command line. Fields are usually a fixed number of character positions in size, but they may also vary.

file

A collection of information. Files may contain data, programs, or other text. You access UNIX system files by name. See **ordinary file, permanent file,** and **executable file**.

file name

A sequence of characters that denotes a file. (In the UNIX system, a slash character (/) cannot be used as part of a file name.)

file system

A collection of files and the structure that links them together. The file system is a hierarchical structure--that is, a ranked system of files. (For more detail, see *Appendix B, File System Organization.*)

filter

A command that reads the standard input, acts on it in some way, and then prints the result as standard output.

final copy

The completed, printed version of a file of text.

foreground

The normal type of program execution. In foreground mode, the shell waits for a command to end before prompting you for another command. In other words, you enter something into the computer and the computer "replies" before you enter something else.

full-duplex

A type of data communication in which a computer system can transmit and receive data simultaneously. Terminals and modems usually have settings for half-duplex (one-way) and full-duplex communication; the UNIX system uses the full-duplex setting.

full path name

A path name that originates at the root directory of the UNIX system and leads to a specific file or directory. Each file and directory in the UNIX system has a unique full path name, sometimes called an absolute path name. See **path name**.

global

A qualifier that indicates the complete or entire file. While normal editor commands commonly act on only the first instance of a pattern in the file, global commands perform the action on all instances in the file.

hardware

The physical machinery of a computer and any associated devices.

hidden character

One of a group of characters within the standard ASCII set, but not normally printed as visible symbols. Control characters, such as backspace and escape, are examples.

home directory

The directory in which you are located when you log in to the UNIX system; also known as your login directory.

input/output

The path by which information enters a computer system (input) and leaves the system (output). An input device that you use is the keyboard and an output device is the terminal monitor.

insert mode

A text editing mode in which you enter (insert) text before the current position in the buffer. See **text input mode**, compare with **append mode** and **command mode**.

interactive

Describes an operating system (such as the UNIX system) that can handle immediate-response communication between you and the computer. In other words, you interact with the computer from moment to moment.

line editor

An editing program in which text is operated upon on a line-by-line basis within a file. Commands for creating, changing, and removing text use line addresses to determine

where in the file the changes are made. Changes can be viewed after they are made by displaying the lines changed. See **text editor**, compare with **screen editor**.

login

The procedure used to gain access to the UNIX operating system.

login directory

See **home directory**.

login name

A string of characters used to identify a user. Your login name is different from other login names.

log off

The procedure used to exit from the UNIX operating system.

metacharacter

One of a group of characters with a special meaning to the **shell**, such as **< > * ? | & $; ( ) \ " ` ' [ ]** .

mode

In general, a particular type of operation (for example, an editor's append mode). In relation to the file system, a mode is an octal number used to determine who can have access to your files and what kind of access they can have. See **permissions**.

modem

A device that connects a terminal and a computer by way of a telephone line. A modem converts digital signals to tones and converts tones back to digital signals, allowing a terminal and a computer to exchange data over standard telephone lines.

multitasking

The ability of an operating system to execute more than one program at a time.

multiuser

The ability of an operating system to support several users on the system at the same time.

nroff

A text formatter available as an add-on to the UNIX system. You can use the **nroff** program to produce a formatted on-line copy or a printed copy of a file. See **text formatter**.

operating system
The software system on a computer under which all other software runs. The UNIX system is an operating system.

option
Special instructions that modify how a command runs. Options are a type of argument that follow a command and are preceded by a minus sign (−). You can specify more than one option for any command given in the UNIX system. For example, in the command **ls −l −a directory**, **−l** and **−a** are options that modify the **ls** command. See **argument**.

ordinary file
A collection of one to several thousand characters. Ordinary files may contain text or other data but are not executable. See **executable file**.

output
Information processed in some fashion by a computer and delivered to you by way of a printer, a terminal, or a similar device.

parameter
Generally, a value that determines the characteristics or behavior of something. In the UNIX system, a type of variable found only on the command line. See **variable**.

parent directory
The directory immediately above a subdirectory or file in the file system organization. The shorthand notation for the parent directory is two dots (..).

parity
A method used by a computer for checking that the data received matches the data sent.

password
A code word known only to you that is called for in the login process. The computer uses the password to verify that you may indeed use the system.

path name
A sequence of directory names separated by the slash character (/) and ending with the name of a file or directory. The path name defines the connection path between some directory and a file.

peripheral device

Auxiliary devices under the control of the main computer, used mostly for input, output, and storage functions. Some examples include terminals, printers, and disk drives.

permanent file

The data stored permanently in the file system structure. To change a permanent file, you must make use of a text editor, which maintains a temporary work space, or buffer, apart from the permanent files. Once changes have been made to the buffer, they must be written to the permanent file to make the changes permanent. See **buffer**.

permissions

Access modes, associated with directories and files, that permit or deny system users the ability to read, write, and/or execute the directories or files. You determine the permissions for your directories or files by changing the mode for each one with the **chmod** command.

pipe

A method of redirecting the output of one command to be the input of another command. It is named for the character (|) that redirects the output. For example, the shell command **who | wc –l** pipes output from the **who** command to the **wc** command, telling you the total number of people logged into your UNIX system.

pipeline

A series of filters separated by the pipe character (|). The output of each filter becomes the input of the next filter in the line. The last filter in the pipeline writes to its standard output. See **filter**.

positional parameters

Variables that hold arguments supplied with a shell procedure. They are placed into variable names, such as **$1**, **$2**, and **$3** when the shell calls for the shell procedure. The name of the shell procedure is positional parameter **$0**. See **variable** and **shell procedure**.

prompt

A cue displayed at your terminal by the shell, telling you that the shell is ready to accept your next request. The prompt can be a character or a series of characters. The UNIX system default prompt is the dollar sign character **($)**.

printer
An output device that prints the data it receives from the computer on paper.

process
Generally a program that is at some stage of execution. In the UNIX system, it also refers to the execution of a computer environment, including contents of memory, register values, name of the current directory, status of files, information recorded at login time, and various other items.

program
The instructions given to a computer on how to do a specific task. Programs are user-executable software.

read-ahead capability
The ability of the UNIX system to read and interpret your input while sending output information to your terminal in response to previous input. The UNIX system separates input from output and processes each correctly.

relative path name
The path name to a file or directory which varies in relation to the directory in which you are currently working.

remote system
A system other than the one on which you are working.

root
The source of all files and directories in the file system, designated by a slash character (/).

screen editor
An editing program in which text is operated on relative to the position of the cursor on a visual display. Commands for entering, changing, and removing text involve moving the cursor to the area to be altered and performing the necessary operation. Changes are viewed on the terminal display as they are made. See **text editor**, compare with **line editor**.

search pattern
See **string**.

search string
See **string**.

secondary prompt
A cue displayed at your terminal by the shell to tell you that the command typed in response to the primary prompt is incomplete. The UNIX system default secondary prompt is the "greater than" character (>).

shell
A UNIX system program that handles the communication between you and the computer. The shell is also known as a command language interpreter because it translates your commands into a language understandable by the computer. The shell accepts commands and causes the appropriate program to be executed.

shell procedure
An executable file that is not a compiled program. A shell procedure calls the shell to read and execute commands contained in a file. This lets you store a sequence of commands in a file for repeated use. It is also called a command file. See **executable file**.

silent character
See **hidden character**.

software
Instructions and programs that tell the computer what to do. Contrast with **hardware**.

source code
The English-language version of a program. The source code must be translated to machine language by a program known as a compiler before the computer can execute the program.

special character
See **metacharacter**.

special file
A file (called a device driver) used as an interface to an input/output device, such as a user terminal, a disk drive, or a line printer.

standard input
An open file that is normally connected directly to the keyboard. Standard input to a command normally goes from the keyboard to this file and then into the shell. You can redirect the standard input to come from another file instead of from the keyboard; use an argument in the form < **file**. Input to the command will then come from the specified file.

standard output

An open file that is normally connected directly to a primary output device, such as a terminal printer or screen. Standard output from the computer normally goes to this file and then to the output device. You can redirect the standard output into another file instead of to the printer or screen; use an argument in the form > **file**. Output will then go to the specified file.

string

Designation for a particular group or pattern of characters, such as a word or phrase, that may contain special characters. In a text editor, a context search interprets the special characters and attempts to match a specified string with an identical string in the editor buffer.

string value

A specified group of characters that is symbolized to the shell by a variable. See **variable**.

subdirectory

A directory pointed to by a directory one level above it in the file system organization; also called a child directory.

system administrator

The person who monitors and controls the computer on which your UNIX system runs; sometimes referred to as a super-user.

terminal

An input/output device connected to a computer system, usually consisting of a keyboard with a video display or a printer. A terminal allows you to give the computer instructions and to receive information in response.

text editor

Software for creating, changing, or removing text with the aid of a computer (known as text processing). Most text editors have two modes--an input mode for typing in text, and a command mode for moving or modifying text. Two examples are the UNIX system editors **ed** and **vi**. See **line editor** and **screen editor**.

text formatter

A program that prepares a file of text for printed output. To make use of a text formatter, your file must also contain some special commands for structuring the final copy. These special commands tell the formatter to justify margins, start

new paragraphs, set up lists and tables, place figures, and so on. Two text formatters available as add-ons to your UNIX system are **nroff** and **troff**.

text input mode
A text editing mode where the text you type is added into the buffer. To execute a command, you must leave the input mode. See **command mode**, compare with **append mode** and insert mode.

timesharing
A method of operation in which several users share a common computer system seemingly simultaneously. The computer interacts with each user in sequence, but the high-speed operation makes it seem that the computer is giving each user its complete attention.

tool
A package of software programs.

troff
A text formatter available as an add-on to the UNIX system. The **troff** program drives a phototypesetter to produce high-quality printed text from a file. See **text formatter**.

tty
Historically, the abbreviation for a teletype terminal. Today, it is generally used to denote a user terminal.

user
Anyone who uses a computer or an operating system.

user-defined
Something determined by the user.

user-defined variable
A shell name given by the user for the value of a string of characters. See **variable**.

UNIX system
A general-purpose, multiuser, interactive, time-sharing operating system developed by AT&T Bell Laboratories. The UNIX system allows limited computer resources to be shared by several users and efficiently organizes the user's interface to a computer system.

utility
Software used to carry out routine functions or to assist a programmer or system user in establishing routine tasks.

variable

A symbol whose value may change within a program or a repetition of a program. In the shell, a variable is a name representing some string of characters (a **string value**). Some variables are normally set only on a command line and are called **parameters** (**positional parameters** and **keyword parameters**). Other variables are simply names to which the user (**user-defined variables**) or the shell itself may assign string values. (Keyword parameters are discussed fully in *UNIX System Shell Commands and Programming;* see description in *Appendix A*.)

video display terminal

A terminal that uses a televisionlike screen (a monitor) to display information. A video display terminal can display information much faster than printing terminals.

visual editor

See **screen editor**.

working directory

See **current directory**.

INDEX

. as current directory, 3-12, 4-9
  as current line in **ed**, 5-16, D-1
  matching any one character in **ed**, 5-57, D-1, D-6
  repeating last command in **vi**, 6-71, E-1
  returning to command mode from text input mode in **ed**, 5-5, D-1
.. parent directory, 3-12, 4-9
? matching any single character, 4-9, 7-3, 7-6, F-1
  searching backward in buffer for text pattern in **vi**, 6-42
  searching forward or backward in buffer in **ed**, 5-22
* matching any number of characters, 4-9, 5-58, 7-3, 7-4, F-1
[] sequence of characters to be matched, 4-9, 5-64, 7-3, 7-8, F-1
– specifying character range within [], 4-9, 7-8
\ removing special character meaning, 2-9, 4-9, 5-62, 6-72, 7-3, 7-11, D-1, E-1, F-1
/ as root directory, 1-6, 3-4, B-3 (see also *Root*)
  as delimiter, 3-4, 5-26, 5-48, D-1
  searching forward in buffer for text pattern, 5-21, 6-42
> redirecting output, 4-11, 4-16, 7-15, F-2
>> redirecting and appending output, 4-13, 4-16, 7-17, F-2
< redirecting input, 4-13, 4-16, 7-14, F-2
| redirecting output of one command as input for another, 4-14, 4-16, 7-19, F-2
; running commands in sequence, 4-16, 7-3, 7-10, F-1
  searching for character in **vi**, 6-22, E-1
$ system command prompt, 2-7
  as last line of buffer in **ed**, 5-17, D-2
  matching end of line in **ed**, 5-60, D-2
erasing a character, 2-7, 5-45
@ erasing entire line of typing in shell or command mode, 2-7
  erasing current line in text input mode in **ed**, 5-44, D-3
  erasing current line in text input mode in **vi**, 6-53, E-2
& background processing, 4-17, 7-3, 7-9, F-1
  pattern substitution in **ed**, 5-63, D-8
! temporarily escaping to shell from **ed** editor, 5-86, D-1
= displaying address of last line of **ed** buffer, 5-16, D-2
^ matching beginning of line in **ed** search, 5-60, D-2
% repeating replacement pattern in **ed**, 5-63, D-3
: beginning line editor command while in **vi**, 6-74, E-1
~ changing lowercase to uppercase, vice versa, in **ed**, 6-71, E-2
'' turning off all special characters, 7-3, 7-11, F-1
"" turning off special characters, 7-3, 7-11, F-1
`` substituting output of command line, 7-23, F-2

A

Acoustic coupler, 2-12
Address
  by line number in **ed**, 5-15
  character string in **ed**, 5-21
  for character string in **ed**, 5-20
  for current line through last line in **ed**, 5-18
  for first line through last line in **ed**, 5-18
  for more than one line in **ed**, 5-23
  relative, 5-19

Answers to exercises, 5-90, 6-88, 7-88
Argument, 3-2, 3-3, 6-54, E-1
Arrays, in C programming language, 4-24
at command, 7-25, C-1, C-3, F-2
  recap of, 7-27

B

Background mode, 4-17, 7-9
banner command, 7-13, C-1, C-4
batch command, 7-23, C-1, C-4, F-2
  recap of, 7-25
Baud rate, 2-4
bbday shell program, 7-40
  recap of, 7-41
bin directory, creating your own, 7-36, F-2
/bin system directory, 1-6, B-3
break statement, 7-75, F-5
Buffers, text editing, 4-2

C

C programming language, 4-23
  control flow, 4-23
  functions and program structure, 4-23
  input and output, 4-24
  pointers and arrays, 4-24
  structures, 4-24
  types, operators, and expressions, 4-23
case...esac construction, 7-72, F-5
cat command, 3-30, C-1, C-4
  recap of, 3-32
cd command, 3-25, C-1, C-4
  recap of, 3-27
CDPATH, shell variable, 7-47, F-3
Changing text in **ed** (see *ed editor*)
Changing text in **vi** (see *vi editor*)
Changing your working directory (**cd**), 3-25
Changing your environment, 4-19, 6-85, 7-80
Changing permissions on files and directories (**chmod**), 3-51, 8-19
Character positioning in **vi** (see *vi editor*)
Character string (see *address*)
chmod command, 3-30, 3-51, 8-19, C-1, F-2
  changing existing permissions with, 3-53
  determining existing permissions with, 3-52
  recap of, 3-56
  symbolic *versus* octal method, 3-35
ch.text shell program, 7-58
  recap of, 7-60
Command line syntax (see *Commands, format of*)
Command mode
  in text editing, 4-3
  in **ed**, 5-5
  in **vi**, 6-8
Commands, 1-9
  advanced, 3-56
  basic, 3-29
  execution of, 1-11
  format of, 3-2
  sequential, 4-16, 7-9
  simultaneous, 4-17
  specifying when to run (**at**), 7-23, F-2
  stopping, 2-9, 7-29
Command output substitution, 7-22
Command prompt, 2-8
Command recaps
  at, 7-27
  batch, 7-25
  cat, 3-32
  cd, 3-27
  chmod, 3-56
  cp, 3-44
  cut, 7-22
  date, 7-20
  diff, 3-59
  echo, 7-4
  grep, 3-61
  kill, 7-30
  lp, 3-41
  ls, 3-25
  mail, 8-8
  mkdir, 3-18
  mv, 3-46
  nohup, 7-31
  pg, 3-36
  pr, 3-38
  ps, 7-29
  pwd, 3-9
  rm, 3-47
  rmdir, 3-29
  sort, 3-64
  spell, 7-15
  uname, 8-12
  uuname, 8-12

uupick, 8-29
uustat, 8-25
uuto, 8-25
wc, 3-50
Communicating with UNIX system users, 8-1
Concatenating and printing (**cat**), 3-29, 3-30
Conditional constructions, 7-66
continue command, 7-76
Control characters, 2-9
Control flow, in C programming language, 4-23
Copying file contents to another file (**cp**), 3-30, 3-41
Copying files from public directory to directory of choice (**uupick**), 4-20, 8-26
Correcting typing errors, 2-8
Counting lines, words, and characters in file (**wc**), 3-30, 3-47
cp command, 3-30, 3-41, C-1, C-4
recap of, 3-44
Creating directories (**mkdir**), 3-16
Creating text in **ed** (see *ed editor*)
Creating text in **vi** (see *vi editor*)
Current directory, 3-8
Cursor (see *vi editor*)
cut command, 7-21, C-1, C-4
recap of, 7-22
Cutting and pasting text electronically, 5-66, E-14

D

date command, 2-19, 7-19, C-1, C-4
recap of, 7-20
Debugging shell programs, 7-77
Deleting text in **ed** (see *ed editor*)
Deleting text in **vi** (see *vi editor*)
/dev system directory, 1-6, B-3
/dev/null, 7-67
diff command, 3-57, C-2, C-5
recap of, 3-59
Differences between files (**diff**), 3-30, 3-57
Directories, 1-6, 3-4
changing (**cd**), 3-16, 3-25
creating (**mkdir**), 3-16
determining permissions for (**chmod**), 3-55
listing contents of (**ls**), 3-16, 3-19
naming, 3-18
organizing, 3-15
removing, 3-16, 3-27
Display commands in **ed** (see *ed editor*)
Displaying a file's contents (**cat, pg, pr**), 3-30
dl shell program, 7-36
recap of, 7-37
Documentation for UNIX system, A-1
descriptions of, A-1
ordering information, A-4
with select codes, A-2

E

echo command, 7-3, C-2, C-5
recap of, 7-4
ed editor
accessing, 5-4, D-1, D-5
addressing current line, 5-16, D-1, D-6
addressing current line through last line, 5-18, D-6
addressing first line through last line, 5-18, D-6
addressing last line in buffer, 5-17
addressing range of lines, 5-24, D-6
answers to exercises, 5-90
appending text, 5-33, D-3, D-5
changing text, 5-37, D-3
command, 4-2, 5-1, C-2, C-5, D-1
commands for using
arranged alphabetically, D-1
arranged by topic, D-5
for getting started, 5-12, D-5
format of, 5-13
copying lines of text, 5-72, D-4, D-9
character string address, 5-21
create commands summarized, 5-39
creating text, 5-5, 5-33, D-3, D-7
current file name, 5-84, D-3, D-9
current line address character, 5-15, D-1, D-6
deleting commands summarized, 5-47, D-7
deleting commands used in text input mode, 5-44, D-7
deleting text, 5-8, 5-41, D-7
deleting current line, 5-8, 5-44, D-5, D-7
deleting last characters typed, 5-45, D-7
display commands summarized, 5-33, D-7
displaying lines of text, 5-6, 5-30, D-7

displaying lines of text preceded by line address number, 5-31, D-7
displaying nonprinting characters, 5-82, D-4, D-9
displaying text, 5-6, 5-30, D-7
ed.hup file, D-9
escaping to shell, 5-86, D-1
exercises, 5-13, 5-29, 5-39, 5-54, 5-67, 5-79, 5-88
getting started, 5-3, D-5
global searches, 5-26, D-3
global substitution, 5-52, D-4
help commands, 5-79
in a shell program, 7-58
inserting text, 5-36, D-4
introduction to, 4-4, 5-1
joining contiguous lines, 5-74, D-4, D-9
last line address character, 5-17, D-2, D-6
line addresses, 5-14, D-1, D-2, D-6
line addressing commands, 5-28, D-6
line addressing symbols, 5-16
line number addressing, 5-15
modes of operation, 4-3
moving around in a file, 5-9, D-5
moving lines of text, 5-69, D-4, D-8
moving text commands summarized, 5-78, D-9
quick reference to commands, D-1
quitting, 5-11, D-4, D-61
reading in contents of file, 5-77, D-4, D-9
recovering from system interrupt, 5-87
relative addressing, 5-19
returning to command mode from text input mode, 5-5, D-1, D-5
saving buffer contents of a file, 5-10
searching for patterns, 5-21, 5-22, D-2, D-5
special characters, 5-56, 5-67, D-1, D-8
special symbols, 5-16
specifying range of lines, 5-24
substituting in range of lines, 5-50, D-4, D-7
substituting text on one line, 5-50, D-4 D-7
substituting on current line, 5-49, D-4, D-7
substituting text, 5-47, D-7
text input, 5-5, 5-33, D-7
text movement commands, 5-9, D-8
undoing last command, 5-43, D-5
writing contents of buffer to file, 5-10
writing lines of text to file, 5-10, 5-75, D-5, D-9
ed.hup file, 5-87, D-9
Electronic communication, 4-20, 8-1 (see *mail, uuto, uupick,* and *mailx*)
enter.name shell program, 7-64
Escape to the shell, 5-86
/etc system directory, 1-8, B-3
Executing a shell program, 7-35
Executing and terminating processes, 7-23
Executing commands in sequence, 4-16, 7-10
Executing commands simultaneously, 4-17, 7-9
Exercises
for line editor (**ed**), 5-13, 5-29, 5-39, 5-54, 5-67, 5-79, 5-88
for screen editor (**vi**), 6-15, 6-45, 6-60, 6-70, 6-84
for shell command language, 7-31
for shell programming language, 7-86
exit command, 7-76
Expressions, in C programming language, 4-23
External security code, 2-16

F

Files, 1-5
accessing and manipulating, 3-29
concatenating and printing contents of (**cat**), 3-29, 3-30
counting lines, words, characters in (**wc**), 3-30, 3-47
determining permissions for (**chmod**), 3-30, 3-51
editing (see *ed editor* and *vi editor*)
identifying differences between (**diff**), 3-57
naming, 3-18
making duplicate copies of (**cp**), 3-30, 3-41
moving and renaming (**mv**), 3-30, 3-44
ordinary, 1-5, 3-4
paging through contents of (**pg**), 3-29, 3-32

printing partially formatted contents of (**pr**), 3-30, 3-36
receiving via **uupick**, 8-26
requesting paper copies of files (**lp**), 3-30, 3-39
removing (**rm**), 3-30, 3-46
searching for patterns in (**grep**), 3-57, 3-59
sending and receiving, 8-17
sending small via **mail**, 8-17
sending large via **uuto**, 8-19
sorting and merging (**sort**), 3-57, 3-62
special, 1-6, 3-4
File system structure, 1-5, 3-4, B-1
major system directories in, 1-6, B-3
for loop, 7-61, F-4
Foreground mode, 4-17
Full duplex, 2-4
Full path names, 3-10
Functions, in C programming language, 4-24

G

gbday shell program, 7-57
recap of, 7-58
General purpose system, 1-2
Generating parser programs (**yacc**), 4-27
Generating programs for lexical tasks (**lex**), 4-27
get.num shell program, 7-43
recap of, 7-44
grep command, 3-37, 3-59, C-2, C-5
recap of, 3-61

H

Half duplex, 2-18
Help commands, 5-79
here document, 7-56, F-3
HOME, shell variable, 4-19, 7-47, 7-83, F-3
Home directory, 3-6

I

IFS, shell variable, 7-47, F-3
Input mode (see *text input mode*)
Interactive computing environment, 1-2
Interrupts, 5-87, 6-77
if...then constructions, 7-66, F-4
if...then...else constructions, 7-67, F-5

K

Kernel, 1-3, 1-4
Keyboard characteristics, 2-4
kill command, 7-29, C-2, C-5, F-2
recap of, 7-30

L

lex command, 4-27, C-2, C-5
/lib system directory, 1-8, B-3
Line addressing in **ed** (see *ed editor*)
Line editor, 1-4, 4-4, 5-1 (also see *ed editor*)
Line positioning in **vi** (see *vi editor*)
Line printers, 3-39
Listing the contents of a directory (**ls**), 3-16, 3-19
Local system (see *Sending small files*)
Logging off, 2-20
Login names, 2-11
Login procedure, 2-13
LOGNAME, shell variable, 4-19
log.time shell program, 7-54
recap of, 7-55
Looping, 7-60
lp command, 3-30, 3-39, C-2, C-5
recap of, 3-41
lpstat command, 3-40, C-2, C-5
ls command, 3-16, 3-19, C-2, C-5
-a option, 3-21
-l option, 3-22
recap of, 3-25

M

mail command, 4-20, 8-4, C-2, C-6
recap of, 8-8
MAIL, shell variable, 7-47, F-3
mailx command, 4-20, 8-29, C-2, C-6
Maintaining programs (**make**), 4-26
make command, 4-26, C-2, C-6
man command, 1-10, C-2, C-6
mkdir command, 3-16, C-2, C-6
recap of, 3-18
mknum shell program, 7-50
recap of, 7-51
Message and file handling (**uucp, mailx**), 8-29
Messages
receiving, 8-12
sending to one person (**mail**), 8-5
sending to remote systems (**uname, uuname**), 8-8
sending to several persons (**mail**), 8-7
Metacharacter, 4-8, 7-3, F-1
that matches all characters, 4-9, 7-4, F-1
that matches one character, 4-9, 7-6, F-1
that matches one of a specific range of characters, 4-9, 7-8, F-1
turning off special meaning of, 2-9, 4-9, 7-11, F-1
turning off special meaning by quoting, 7-11, F-1
Modes of editor operation, 4-3
Moving and renaming files (**mv**), 3-30, 3-44
Moving text in **ed** (see *ed editor*)
Multitasking, 1-2
mv command, 3-30, 3-44, C-2, C-6
recap of, 3-46
mv.ex shell program, 7-70
recap of, 7-71
mv.file shell program, 7-62
recap of, 7-63

N

nohup command, 4-18, 7-30, C-2, C-6, F-2
recap of, 7-31
No interrupt command, 7-30
num.please shell program, 7-49
recap of, 7-52

O

Obtaining status of running processes, 7-28
On-line operation, 2-4
Operators, in C programming language, 4-23
Options, 3-2, 3-3

P

Paging through file contents (**pg**), 3-29, 2-32
Paper-printing terminal, 2-2, 4-4
Paragraph positioning in **vi** (see *vi editor*)
Parameters
positional, 7-39, F-2
in command lines, 5-13, 5-14, D-1
with special meaning, 7-43
Parity, 2-4
Password, 2-14
PATH, shell variable, 4-19, 7-47, 7-83, F-3
Path names, 3-9
full or absolute, 3-9
relative, 3-12
pg command, 3-29, 3-32, C-2, C-7
recap of, 3-36
Permissions (see *chmod*)
Pipes (|), 4-14, 7-19
Pointers, in C programming language, 4-24
Positional parameters (see *Parameters*)
Positioning in file in **vi** (see *vi editor*)
Positioning in window in **vi** (see *vi editor*)
pr command, 3-30, 3-36, C-2, C-7
recap of, 3-38
Printing partially formatted file contents (**pr**), 3-30, 3-36
Problems when logging in, 2-17
Processes, executing and terminating, 7-23
.profile, 4-19, 6-85, 7-80, 7-81
Program structure, in C programming language, 4-24
Programming
shell, 4-21, 7-32
system, 4-21
Programming languages, 4-23, 4-24
Protecting your files (**chmod**), 3-50

ps command, 7-28, C-2, C-7, F-2
recap of, 7-29
PS1, PS2, shell variables, 7-48, 7-85, F-3
pwd command, 3-6, C-2, C-7
recap of, 3-9

Q

Quick reference
to **ed** commands, D-1
to **vi** commands, E-1
Quoting, 7-11, F-1

R

Recaps, for commands
(see *Command recaps*)
Receiving files (**uupick**), 8-26
Receiving messages (**mail**), 8-12
Redirection
and appending output, 4-13, 7-17, F-2
of input, 4-13, 7-14, F-1
of output, 4-11, 7-15, F-2
with pipes, 4-14, 7-19, F-2
Relative path names, 3-12
Remote job entry (**RJE**), 4-26
Remote operation, 2-4
Remote system, sending mail to, 8-8
Removing files (**rm**), 3-30, 3-46
Removing directories (**rmdir**), 3-16, 3-27
Requesting a paper copy of a file
(**lp**), 3-30, 3-39
RJE, 4-26
rm command, 3-30, 3-46, C-2, C-7
recap of, 3-47
rmdir command, 3-16, 3-27, C-2, C-7
recap of, 3-29
Root, 1-6, 3-4, 3-7, 3-9
Running multiple programs, 4-16

S

SCCS, 4-25
Screen editor, 4-5, 6-1 (see also *vi editor*)
Scrolling in **vi** (see *vi editor*)
search shell program, 7-68
recap of, 7-69
Searching a file for a pattern
(**grep**), 3-30, 3-59
Sending files
large (**uuto**), 8-19
small (**mail**), 8-17
via public UNIX-to-UNIX system
(**uuto, uustat**), 8-21
Sending messages
basics of, 8-4
to one person, 8-5
to several people, 8-7
to remote systems, 8-8
Sentence positioning in **vi** (see *vi editor*)
set.term shell program, 7-73
recap of, 7-75
sh command, 7-35, C-2, F-2
Shell, 1-8
answers to command language exercises, 7-88
command language, 4-6, 7-3, F-1
command language exercises, 7-31
executing and terminating processes in the, 7-23, F-2
programming language, 4-21, 7-32
redirecting input/output in the
(see *Redirection*)
special characters in the
(see *Metacharacters*)
variables in the, 7-38, F-2
Shell program recaps
bbday, 7-41
ch.text, 7-60
dl, 7-37
gbday, 7-58
get.num, 7-44
log.time, 7-55
mknum, 7-51
mv.ex, 7-71
mv.file, 7-63
num.please, 7-52
search, 7-69
set.term, 7-75
show.param, 7-44
t, 7-53
tail, 7-83
whoson, 7-43
Shell programming
answers to exercises, 7-88
constructions, 7-55, F-3
creating a simple program, 7-34
executing programs, 7-35
exercises, 7-86
language, 7-32
variables, 7-38, F-2
Shell scripts (see *Shell programming*)
Shell shorthand (see *Metacharacters*)

show.param shell program, 7-44
recap of, 7-45
Software development, 4-25
generating parser programs (**yacc**), 4-27
generating programs for lexical tasks (**lex**), 4-27
maintaining programs (**make**), 4-26
remote job entry (**RJE**), 4-26
Source Code Control System (**SCCS**), 4-25
sort command, 3-57, 3-62, C-2, C-7
recap of, 3-64
Sorting and merging files (**sort**), 3-57, 3-63
Source Code Control System (**SCCS**), 4-25
Special characters (see *Metacharacters*)
Special characters in **ed** (see *ed editor*)
spell command, 7-14, C-3, C-8
recap of, 7-15
Standard input and output, 4-9
Standard input/output (I/O) library, 4-24
Stopping commands, 2-9, 7-20
Stopping execution of at or batch job, 7-22, F-2
Structures, in C programming language, 4-24
stty command, 2-18, 7-82, C-3, C-8
Substituting text in **ed** (see *ed editor*)

T

t shell program, 7-52
recap of, 7-53
tail shell program, 7-82
recap of, 7-83
tee command, 7-79
TERM, shell variable, 7-79 F-3
Terminal, 2-2
configuration for **vi**, 6-5, E-8
keyboard characteristics, 2-4
optional settings, 7-82
paper-printing, 2-2, 4-4, 5-3
required settings, 2-3
typing conventions, 2-6
setting automatic carriage return in vi, 6-86
typing speed, 2-9
video display, 2-2, 4-4, 5-2, 6-1
Terminating active processes, 7-29
test command for loops, 7-69
Text editing, 4-1
buffers, 4-2
line-oriented, 4-4, 5-1
modes of operation, 4-3
screen-oriented, 4-5, 6-1
Text editors (see *line editor, ed editor, screen editor, vi editor*)
Text input mode, 4-3
in **ed**, 5-5, 5-33, D-7
in **vi**, 6-12, E-12
Timesharing, 1-2
/tmp system directory, 1-8, B-3
Turn off meaning of special characters by quoting, 7-3, 7-11, F-3
Tutorial
for electronic communication, 8-1
for line editor (**ed**), 5-1
for shell command and programming languages, 7-1
for visual editor (**vi**), 6-1
Types, in C programming language, 4-23
Typing
conventions, 2-6
nonprinting characters, 6-67
speed, 2-9

U

uname command, 8-8, C-2, C-8
recap of, 8-12
Unconditional control statements, 7-64
UNIX system description, 1-1
/usr system directory, 1-8, B-4
uucp command, 8-29, C-3, C-8
uuname command, 8-8, C-3, C-8
recap of, 8-12
uupick command, 4-20, 8-26, C-3, C-8
recap of, 8-29
uustat command, 8-21, C-3, C-8
recap of, 8-25
uuto command, 4-20, 8-21, C-3, C-8
recap of, 8-25

V

Variables, 7-38, F-2
assigning values to, 7-48, F-3
assigning name values with positional parameters, 7-54
assigning name values by **Read** command, 7-48

assigning name values with
command output, 7-52
names, 7-46, F-3
positional parameters, 7-39, F-2
used by the shell, 7-46, F-3
vi editor
accessing, 6-7, E-1, E-8
adding file to buffer, 6-77
adding text, 6-13, E-2, E-8
appending text, 6-47
answers to exercises, 6-88
changing lowercase characters to
uppercase characters, 6-73, E-2
changing text, 6-60, 6-63, E-13
changing uppercase characters to
lowercase characters, 6-73, E-2
changing your environment, 6-85
character positioning, 6-18, E-8
clearing and redrawing the window,
6-73, E-5, E-14
command, 6-1, C-3, C-8, E-1
commands for using
alphabetically arranged, E-1
arranged by topic, E-8
for getting started, 6-5, 6-15, E-8
copying
or moving text using registers, 6-69
copying text, 6-68
creating text, 6-8, 6-46, E-12
cutting and pasting text, 6-66, E-14
deleting commands in command
mode, 6-54, E-12
deleting commands in text input
mode, 6-51, E-12
deleting rest of buffer, 6-77, E-3
deleting text, 6-12, 6-51, E-3, E-12
deleting text objects, 6-55, E-13
escaping to shell, 6-75, E-15
exercises, 6-15, 6-45, 6-50, 6-60, 6-70,
6-84
finding line number, 6-76
fixing typos, 6-67
getting started in, 6-5, E-8
global changes, 6-77
hidden characters (see *nonprinting characters in vi*)
inserting text, 6-47
moving to specified line, 6-40
joining two lines, 6-72, E-14
leaving append mode, 6-9
line editor commands in,
6-74, 6-79, E-1, E-15
line numbers, 6-40
line positioning, 6-23, E-9
moving cursor, 6-9, E-2, E-3
moving text, 6-66
nonprinting characters, 6-71, E-1,
E-14
positioning cursor at end or
beginning of line, 6-20, E-9
positioning cursor by paragraphs,
6-29, E-10
positioning cursor by sentences,
6-27, E-10
positioning cursor in the file,
6-34, E-11
positioning cursor in window, 6-16,
6-30, E-8, E-9
positioning cursor to right or left,
6-18
positioning on a numbered line,
6-40, E-11
quick reference to commands, E-1
quitting, 6-14, 6-80, E-16
repeating last command, 6-71, E-14
replacing text, 6-61, E-6
restoring file after system interrupt,
6-82, E-16
scrolling text, 6-35, E-11
searching for character on line, 6-21,
E-2, E-11
searching for character pattern, 6-41,
E-2, E-11
setting automatic carriage
return, 6-86
special commands, 6-71, E-14
special options, 6-77, E-16
substituting text, 6-62
terminal configuration, 6-5, E-8
undoing last command, 6-53, E-6
word positioning, 6-25, E-9, E-10
writing text to new file, 6-75,
E-6, E-7
Video display terminal
(VDT), 2-2, 4-4, 6-1
Visual editor (see *screen editor*)

W

wc command, 3-30, 3-47, C-3, C-9
recap of, 3-50
while loop, 7-64, F-4
who command, 2-19, C-3, C-9
whoson shell program, 7-42
recap of, 7-43
Window positioning in **vi** (see *vi editor*)
Word positioning in **vi** (see *vi editor*)
Working directory, 3-6

Y

yacc command, 4-27, C-3, C-9